Universita' degli Studi di Cagliari
Dottorato di Ricerca
in
Ingegneria Elettronica ed Informatica
XVIII Ciclo

Experimenting Abstraction Techniques to manage Complexity in Agent Systems and Planning

Tesi di Dottorato - PhD Thesis by

Giancarlo Cherchi

Responsabile Scientifico - Scientific Advisor
Giuliano Armano

Coordinatore - Coordinator
Massimo Vanzi

February 2006

Dedication

This thesis is dedicated to my mom and dad - Elisa and Flavio - for the importance given to my education, being encouraging and enthusiastic about all my career, and being always "with me", in every sense, during all my life journey.

Abstract

The increasing complexity of modern computer and information processing systems goes together with an increasing complexity of their applications. New systems are required, which are able to operate robustly in rapidly changing and unpredictable environments, where there is a significant possibility that actions can fail. An emergent research area in this field is that of intelligent agent systems, which are able to exhibit autonomous and flexible behaviour. In particular, multiagent systems offer a promising and innovative way to manage and use distributed, large-scale, dynamic, open, and heterogeneous computing and information systems.

This thesis faces with several issues surrounding the development of multiagent systems. To manage the inherent complexity of such applications, abstraction-based techniques are investigated. Abstraction allows to concentrate on the most important aspects of a problem first – according to a "divide et impera" stategy – instead of managing at the same time all details.

In particular, a novel multiagent architecture is described, designed to support the implementation of applications aimed at: (i) retrieving heterogeneous data spread among different sources, (ii) filtering and organizing them according to personal interests explicitly stated by each user, and (iii) providing adaptation techniques to improve and refine throughout time the profile of each selected user. The architecture has been called *PACMAS*, which stands for "Personalized Adaptive Cooperative MultiAgent System". The effectivenes of the architecture is highlighted by two relevant case studies that have been implemented exploiting the *PACMAS* architecture: the first one is focused on giving a support to undergraduate and graduate students in their university activities; the second one is devoted to create press-reviews from online newspapers through the classification of newspaper articles. Successfull tests, performed on the developed applications, demonstrate the effectiveness of the architecure.

In order to equip the involved agents with pro-active capabilities, advanced planning algorithms are needed. To this aim, the parametric system *HW[ ]* has been devised and implemented to perform planning by abstraction that is an effective approach for implementing the planning capabilities of an intelligent agent. In particular, *HW[ ]* is able to improve the per-

formances of a *generic* planner in complex domains through abstraction techniques, and to solve problems of high complexity that cannot be solved with traditional approaches.

Since finding good abstractions by hand is a long and difficult process, a system devoted to automatically generate abstraction hierarchies, called *DHG*, has been devised to support *HW[ ]*. Experimental results are encouraging and highlight that abstraction is useful for improving the performances of classical planners. Moreover, a direct comparison between the performances of automatically-generated versus hand-coded abstraction hierarchies demonstrates the validity of the approach.

List of Publications

During the research activity reported in this thesis, the following articles have been published:

- G. Armano, G. Cherchi, S. Fernandez, and E. Vargiu. Integrating Abstraction Techniques and EBL Control Rules to Perform Automated Planning. In *Proceedings of PLANSIG 2005, the 24th Annual Workshop of the UK Planning and Scheduling Special Interst Group*, City University, London, UK 15-16 Dicembre 2005.

- G. Armano, G. Cherchi, A. Manconi, and E. Vargiu. PACMAS: A Personalized, Adaptive, and Cooperative MultiAgent System Architecture. In Proceedings of *WOA 2005 (Simulazione e Analisi Formale di Sistemi Complessi)*, Camerino 14-16 Novembre 2005.

- G. Cherchi, D. Deledda, A. Manconi, and E. Vargiu, Text Categorization Using a Personalized, Adaptive, and Cooperative MultiAgent System. In Proceedings of *WOA 2005 (Simulazione e Analisi Formale di Sistemi Complessi)*, Camerino 14-16 Novembre 2005.

- G. Armano, P. Baroni, G. Cherchi, M. Colombetti, A. Gerevini, M. Mari, A. Poggi, C. Santoro, E. Tramontana, and M. Verdicchio. ANEMONE - A Network of Multi-Agent Platforms for Academic Communities. In *Proceedings of WOA 2005 (Simulazione e Analisi Formale di Sistemi Complessi)*, Camerino 14-16 Novembre 2005.

- G. Armano, G. Cherchi, and E. Vargiu. DHG: A System for Generating Macro-Operators from Static Domain Analysis. In *Proceedings of International Conference on Artificial Intelligence and Applications (AIA'05).*

- G. Armano, G. Cherchi, and E. Vargiu. Automatic Generation of Macro-Operators from Static Domain Analysis. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004)*, Valencia (Spain), August, 2004.

- G. Armano, G. Cherchi, and E. Vargiu. A Critical Look at the Abstraction Based on Macro-Operators. In *Advances in Artificial Intelligence, 9th Congress of the Italian Association for Artificial Intelligence*, Perugia (Italy), September 2004.

- G. Armano, G. Cherchi, and E. Vargiu. Planning by Abstraction Using HW[]. *AI*IA 2003: Advances in Artificial Intelligence*, LNAI 2829:349–361, 2003.

- G. Armano, G. Cherchi, and E. Vargiu. Generating Abstractions from Static Domain Analysis. *In Proceedings of WOA 2003 (Dagli Oggetti agli Agenti, Sistemi Intelligenti e Computazione Pervasiva)*, Villasimius (CA) 10-11 September 2003.

- G. Armano, G. Cherchi, and E. Vargiu. A Parametric Hierarchical Planner for Experimenting Abstraction Techniques. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, Acapulco (Mexico), August, 2003.

- G. Armano, G. Cherchi, and E. Vargiu. HW[ ]: A Parametric System for Planning by Abstraction. *PLANET News*, 6:5-10, 2003.

- G. Armano, G. Cherchi, and E. Vargiu. An Extension to PDDL for Hierarchical Planning. In *Proceedings of Workshop on PDDL, International Conference on Planning and Scheduling (ICAPS'03)*, Trento (Italy), June, 2003.

- G. Armano, G. Cherchi, and E. Vargiu. Experimenting the Performance of Abstraction Mechanisms through a Parametric Hierarchical Planner. In *Proceedings of IASTED International Conference on Artificial Intelligence and Applications (AIA'03)*, Innsbruck, Austria, February 10-13, 2003.

Acknowledgements

Although a few words do not do justice to their contribution, I would like to thank the following people for making this work possible. First, I would like to acknowledge my thesis advisor Giuliano Armano. He has constantly provided excellent encouragement and guidance on my research work and academic matters. His attitude toward my shortcomings during the course of this research has been extremely constructive with prompt suggestions about what would be a way to improve or a better way to proceed in my work. A special mention to my colleague Eloisa Vargiu, with whom I share a lot of research experiences. Finally, I wish to thank Dario Deledda and Arcadia Design for their active organization of projects that gave me the possibility to put in practice my research theories.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We are in the Information Age. The amount and availability of electronically stored information is continually growing. The last few years have seen an information explosion due to significant cost reductions in data storage technologies, and telecommunications developments that have enhanced the inter-connectivity and efficiency of data transfer among sites. Modern computing platforms as well as information environments are large, open, heterogeneous, and distributed. Computers are no longer stand-alone systems, but have become tightly connected both with each other and their users. The increasing complexity of computer and informatics systems goes together with an increasing complexity of their applications.

1.1 Motivation

Traditional techniques reveal inadequate to cope with modern applications. In fact, present application environments are complex, dynamic, and partially accessible. The Internet is perhaps the most prominent example of such an environment. Therefore, new systems are required, which are able to operate robustly in rapidly changing and unpredictable environments, where there is a significant possibility that actions can fail.

Software engineering has evolved together with the complexity of applications. In particular, computer programming has evolved from simple low-level programs to progressively more complex software systems: it has progressed through sub-routines, procedures and functions, abstract data types, objects, components, and is flowing naturally towards agents.

Agent-oriented programming is a modern paradigm that provides a technology for developing system that decide for themselves what they need to do in order to satisfy their design objectives. In other words, autonomy and flexibility become the central aspects in this context. In particular, multiagent systems offer a promising and innovative way to understand, manage, and use distributed, large-scale, dynamic, open, and heterogeneous comput-

ing and information systems.

There exist many potential applications for multiagent systems; for example:

- electronic commerce, where agents purchase and sell goods on behalf of their users;
- real-time monitoring and management of telecommunication networks, where agents are responsible for call forwarding and signal switching;
- modelling and optimization of transportation systems;
- information handling in information environments like the Internet, where agents are responsible - for example - for information gathering and filtering;
- improving the flow of urban or air traffic, where agents are responsible for appropriately interpreting data arising at different sensor stations;
- automated meeting scheduling, where agents fix meeting details like location, time and agenda, to their users;
- electronic entertainment and interactive computer games, where agents equipped with different characters play against each other or against humans.

These applications have in common that they are inherently distributed in data and information to be processed, and they are inherently complex in the sense that they are too large to be solved by a single, centralized system beacuse of limitations available at a given level of hardware or software technology.

1.2 Objective and Approach

The main objective of this research work is to provide methodologies that are effective to develop such applications. In particular, they must be able to manage the inherent complexity of the corresponding environments. We need applications "user-oriented": they must be personalized to users' needs, and able to adapt themselves to the evolving environment.

We said that a succesfull approach is represented by multiagent technology. In fact, multiagent systems can offer several desiderable properties, such as:

- speed up and efficiency: agents can operate asynchronously and in parallel, and this can result in an increased overall speed;

- robustness and reliability: the failure of one or several agents does not necessarily make the overall system useless, because other agents already available in the system may take over their part;
- scalability and flexibility: the system can be adopted to an increased problem size by adding new agents, and this does not necessarily affect the operationality of the other agents;
- costs: it may be much more cost-effective than a centralized system, since it could be composed of simple subsystems of low unit cost;
- development and reusability: individual agents can be developed separately by specialists, the overall system can be tested and mantained more easily, and it may be possible to reconfigure and reuse agents.

On the other hand, there are several issues to be taken into account while developing such systems. In particular, agents should be able to communicate with other agents and the user, to adapt themselves to user's needs. Therefore, suitable *architecture* are needed. Moreover, agents need to act in their environments, so that effective *planning* algorithm are also needed.

The proposed approach exploits *abstraction* techniques to manage the inherent complexity of such systems. Abstraction allows concentrating on the most important aspects of a problem first, according to a "divide et impera" stategy, instead of managing at the same time all the detailed aspects of the problem itself.

Abstraction is therefore used to model a multiagent system through a layered architecture that allows to manage complexity at different levels of granularity, thus separating the different aspects of the problem.

Abstraction is also used to manage the complexity of agent's planning capabilities, by exploiting a hierarchy of abstraction spaces to improve the performances of the search process.

1.3 Contributions

My research work has mainly focused on: *intelligent agents*, with particular emphasis on the study and realisation of multiagent systems able to support the user by adapting to her/his *personal* needs; *automated planning*, with particular emphasis on the study and realisation of abstraction-based algorithms to implement effective agent's pro-active behaviour. This thesis gives therefore contributions in two areas: agent systems and planning.

1.3.1 Agent Systems

Research in this area has lead to a novel multi-agent architecture designed to support the implementation of applications aimed at:

1. retrieving heterogeneous data spread among different sources (e.g., generic html pages, news, blogs, forums, and databases);

2. filtering and organizing them according to personal interests explicitly stated by each user;

3. providing adaptation techniques to improve and refine throughout time the profile of each selected user.

The architecture has been called *PACMAS*, which stands for "Personalized Adaptive Cooperative MultiAgent System".

Upon the PACMAS architecture, two applications have been developed as case studies: an e-service devoted to support undergraduate and graduate students in their university activities, and a classification system devoted to create personalized press-reviews from online newspapers.

The first one is able to retrieve relevant information from heterogeneous sources (e.g.: files, forums, databases, professor homepages, department web pages, etc.), and then filter, organize, and present it to the user, according to her/his personal needs and preferences.

The second one is able to extract from web sites of online newspapers the articles deemed relevant for a specific user, and to improve user's profile through learning algorithms. The selection procedure is made by suitable classifying algorithms.

1.3.2 Planning

Research in this area has lead to the realisation of two systems: the Hierarchical Wrapper *HW[ ]* and the Domain Hierarchy Generator *DHG*.

The parametric system *HW[ ]* is able to improve the performances of a *generic* planner in complex domains through abstraction techniques, and to solve problem of high complexity that cannot be solved with traditional approaches.

DHG is a supporting system able to automatically generate abstraction hierarchies to be used by HW[ ] to plan hierarchically.

1.4 Outline

This thesis is divided into three parts.

Part I is organized into three chapters, which describe agent systems, planning, and abstraction. Chapter 2 presents the basic issues surrounding the design and implementation of intelligent agents. Chapter 3 focuses on the aspects regarding the pro-active capability of agents. Chapter 4 illustrates the abstraction techniques that can be exploited to improve the performances of automated classical planners, which allow implementing the planning capabilities of intelligent agents.

Part II is organized into two chapters, which describe the proposed Personalize Adaptive Cooperative MultiAgent System that is one of the major contributes of this work, together with two relevant case studies. Chapter 5 describes the PACMAS architecture, designed to support the implementation of applications aimed at: (i) retrieving heterogeneous data spread among different sources, (ii) filtering and organizing them according to personal interests explicitly stated by each user, and (iii) providing adaptation techniques to improve and refine throughout time the profile of each selected user. Chapter 6 presents two relevant case studies that have been implemented exploiting the PACMAS architecture: the first one being focused on giving a support to undergraduate and graduate students in their university activities; the second one being devoted to create press-reviews from online newspapers through the classification of newspaper articles.

Part III is divided in two chapters, and is aimed at presenting two further contributes of this thesis: the parametric system *HW[ ]* for planning by abstraction, and the *DHG* system for automatically generate abstraction hierarchies. Chapter 7 presents a novel approach for implementing the planning capabilities of an intelligent agent, which exploits the parametric system *HW[ ]* that has been devised and implemented to perform planning by abstraction. Chapter 8 describes a novel system, called *DHG*, devised to automatically generate abstraction hierarchies, by macro-operator extraction after a static domain analysis.

The final chapter 9 draws conclusions and outlines some possible future work in this research area.

Part I

Agents Systems, Planning and Abstraction

Chapter 2

Agent Systems

Intelligent software agents are a popular research topic in various fields, such as psychology, sociology and computer science. They are particularly studied in the discipline of Artificial Intelligence. The increasingly interest in agent-based systems has introduced a kind of new paradigm for software engineering, i.e., the agent-oriented programming (AOP). Researchers in this area claim that computer programming has progressed through sub-routines, procedures and functions, abstract data types, objects, components, and it is flowing naturally towards agents.

The most important theoretical and practical issues associated with the design and implementation of intelligent agents can be organized into three areas: agent theory, agent architectures, and agent languages.

Agent *theory* is concerned with the question of "what an agent is", and the use of mathematical formalisms for representing and reasoning about the properties of agents.

Agent *architectures* can be considered as software engineering models of agents; researchers in this area are primarily concerned with the problem of designing software or hardware systems that will satisfy the properties specified by agent theorists.

Agent *languages* are software systems for programming and experimenting with agents; these languages may embody principles proposed by theorists.

This chapter introduces the basic issues surrounding the design and implementation of intelligent agents. Section 2.2 begins by motivating the idea of an agent, presents a definition of agents and intelligent agents, defines the properties of the environment in which agents act, and then discusses the relationship between agents and other software paradigms (in particular, objects and expert systems).

Section 2.3 describes agent architectures, first by an abstract point of view, then by a more concrete point of view, focusing on the major approaches for building agents.

In general, there is a distinction between the *agent* level, the "micro", and the *group* level, the "macro" (see [MCd96]). The question how agent-level (*individual* activity) and group-level (*societal* rules and structures) are related to each other is known as the micro-macro problem in sociology. According to this distinction, section 2.3.2 discusses four major *micro-architectures* for building agents (logic based, reactive, layered, and belief-desire-intention - BDI), while section 2.3.3 describes *macro-architectures*, through which agents can operate and interact with each others.

Finally, section 2.4 introduces some prototypical programming languages for agent systems.

2.1 Introduction

The interest about agents systems has undergone a strong growth since the mid-1980s. Since then, many new discussion topics have been proposed, including the definition of agent itself, as well as the most important properties that characterize agent-based systems. It could be surprising that, despite the term *agent* is widely used by many people working in the AI community, nowadays a single universally accepted definition does not exist. Strangely enough, it seems that the question of what exactly an agent is, has only very recently been addressed seriously. Carl Hewitt, during an international workshop on distributed AI, remarked that the question *"what is an agent?"* is embarrassing for the agent community in just the same way that the question *"what is intelligence?"* is embarrassing for the mainstream AI community.

2.2 Agent Theory

Since the term *agent* is currently used by many parties in many different ways, it has became difficult for users to make a good estimation of what the possibilities of the agent technology are. Moreover, given the multiplicity of roles agents can play, a rock-solid formal definition of the concept *agent* is quite impossible and even very impractical. However, a certain agreement concerning the general characteristics that agents system should possess, has been reached. Together these characteristics give a global impression of what an agent "is".

2.2.1 What is an agent?

Suprisingly, there is no agreement on what an agent exactly is. Despite of there is a general consensus that *autonomy* is the central notion in agent-based systems, there is, however, little agreement beyond this. Part of

the difficulty is that properties associated with agents are often of different importance for different domains. Nevertheless, to avoid that the term agent will lose all meanings, we retain the definition proposed by Wooldridge [RN95]:

> *An agent is a computer system (hardware and/or software) that is* situated *in some environment, and that is capable of* autonomous *actions in this environment to meet its design objectives.*

Like agency itself, autonomy is a somewhat tricky concept to be expressed precisely. Anyway, we mean that agents are said to be autonomous if they are able to act without the intervention of humans or other systems: they have control both over their internal state, and over their behaviour.

Figure 2.1: Abstract view of an agent in its environment.

Figure 2.2.1 gives a general abstract view of an agent. Note that an agent is strictly related to the environment in which it acts: the agent takes sensory input from the environment, and produces as output actions that affect it.

According to the previous definition, many systems can be considered as agents (although not intelligent). In principle, any control systems can be viewed an agent. Common examples of such systems are a thermostat and UNIX daemons. In fact, both of them monitor the environment, and perform action to modify it. Thermostats have a sensor for detecting room temperature, and produce two possibile outputs through two different signals: one that indicates that the temperature is too low, another one indicating that temperature is OK. Thus, the action available to the thermostat are "heating on" or "heating off". Many software daemons monitor a software environment (e.g. system events or user inputs) and performs actions (e.g. displaying dialogs, executing a program). Of course, we do not think thermostats as agents, and certainly not as *intelligent* agents! So, when can we consider an agent to be intelligent? Similarly to the question "What is intelligence?", this is not an easy question to answer.

According to Wooldridge [Woo02], an *intelligent agent* is a computer system capable of *flexible* autonomous action in order to meet its design objectives. Flexibility means three things:

- **pro-activity:** intelligent agents are able to exhibit goal-directed behaviour by taking the initiative, in order to satisfy their design objectives;
- **reactivity:** intelligent agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives;
- **social ability:** intelligent agents are capable of interacting with other agents (and possibily humans) in order to satisfy their design objectives.

These properties are more demanding than they might at first appear, and they will be described in more detail in section 2.2.3.

2.2.2 Environments

Agents act in an environment. Often, they are said to be *situated* in a precise environment. The environment can be either real (like a robot world) or a simulation (like a computer game). It is clear that actions and perceptions will differ depending on the specific case. In case of software agents, for example, they could be procedure calls or software events.

The main problem facing an agent is that of deciding which of its actions it should perform in order to satisfy its design objectives. In other words, agents are a kind of *decision-making* systems that are embedded in an environment. The complexity of the decision-making process can be affected by a number of different environmental properties. Therefore, it is important to define a classification of environmental properties. Let us briefly recall the classification suggested by Russel and Norvig [RN95] that is the most commonly accepted in the field of Artificial Intelligence research.

Accessible versus inaccessible

An accessible environment is one in which the agent can obtain complete, accurate, up-to-date information about the environment's state. Most moderately complex environments (including, for example, the everyday physic world and the Internet) are inaccessible in this sense. The more accessible an environment is, the simpler it is to build agents to operate in it.

Deterministic versus non-deterministic

A deterministic environment is one in which any action has a single guaranteed effect. In other words, there is no uncertainty about the state that

will result from performing an action. The physical world can be condidered as non-deterministic. Non-deterministic environments present greater problems for the agent designer.

Episodic versus non-episodic

In an episodic environment, the performance of an agent is dependent on a number of discrete episodes, with no link between the performance of an agent in different scenarios. Episodic environments are simpler from the agent developer's perspective because the agent can decide what action to perform based only on the current episode. Therefore, it needs not to reason about the interactions between the current episode and future episodes.

Static versus dynamic

A static environment is one that can be assumed to remain unchanged except by the actions of the agent itself. In contrast, a dynamic environment is one that has other processes operating on it, and which hence changes in ways beyond the agent's control. The physical world is a highly dynamic environment, as is the Internet.

Discrete versus continuous

An environment is discrete if there are a fixed, finite number of actions and percepts in it. Russell and Norvig give a chess game as an example of a discrete environment, and taxi driving as an example of a continuous one.

It is worth noting that if an environment is sufficiently complex, then the fact that it is actually deterministic is not much help: to all intents and purposes, it may as well be non-deterministic. The most complex general class of environments are those that are inaccessible, non-deterministic, non-episodic, dynamic, and continuous.

2.2.3 Agent Properties

As mentioned above, intelligent agents must exhibit flexible behaviour. In the literature, two different definitions of flexible behaviour have been given, depending on the characteristics that agents possess: the *weak* notion and the *strong* notion.

Weak notion

According to the weak notion, an agent is a hardware- or software-based computer system that enjoys the following properties [1]:

Autonomy. Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.

Proactivity. Agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative and not driven solely by events. To this end, is fundamental the ability of *planning*, i.e. finding sequences of actions (i.e. *plans*) that solve some problems (defined in term of an *initial state* of the world and a *goal* to be reached), which will be described in chapter 3.

Note that building purely goal-directed systems is not particularly hard. We want agents to do things for us, which will attempt to achieve their goal systematically by using suitable complex procedures. On the other hand, we do not want agents to continue blindly execute a procedures in attempt to achieve a goal, either when it is clear that the procedure will not work or when the goal is no longer valid. In such circumnstances, we want agents to react to the new situation, in a timely fashion. Building purely reactive systems is also not difficult. What turns out to be hard is building a system that achieves a *balance* between goal-directed and reactive behaviour.

Reactivity. A reactive system is one that maintains an ongoing interaction with its environment, responding to changes that occur in it, in time for the response to be useful. Since real word environments (as well as many interesting simulated environments) are dynamic, agents must be able to perceive their environment (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it.

Social Ability. Agents interact with other agents (and possibly humans) via some kind of agent-communication language. The interation level depends on the specific system: it ranges from a simple communication to complex forms of cooperation or competition. In a sense, social ability is trivial: every day, millions of computers exchange information with both humans and other computers. But the ability of exchange bit stream is not social ability! Agent systems consider a type of social ability much

[1] The fact that an agent should possess most, if not all of these characteristics, is something that most scientists have agreed upon at this moment.

more complex, which usually refers to concepts including cooperation and negotiation: some goals can only be achieved with the cooperation of other agents.

Strong Notion

For some researchers, the term agent has a stronger and more specific meaning than that sketched out above. These researchers generally mean an agent to be a computer system that, in addition to having the properties as they were previously identified, is either conceptualised or implemented using concepts that are more usually applied to humans. For example, it is quite common in AI to characterise an agent using mentalistic notions, such as knowledge, belief, intention, and obligation. Some AI researchers have even gone further, and considered "emotional" agents.

Agents that fit the strong notion of agent usually have one or more of the following characteristics:

mobility: is the ability of an agent to *move* around an electronic network, retaining its current state;

veracity: is the assumption that an agent will not knowingly communicate false information;

benevolence: is 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: is (crudely) 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;

learning/adaptation: is the ability of an agent to improve its performance over time, as well as being able to adjust itself to the habits, working methods and preferences of its user;

collaboration: an agent should not unthinkingly accept (and execute) instructions, but should take into account that the human user makes mistakes (e.g. give an order that contains conflicting goals), omits important information and/or provides ambiguous information. For instance, an agent should check things by asking questions to the user, or use a built-up user model to solve problems like these. An agent should even be allowed to refuse to execute certain tasks, because (for instance) they would put an unacceptable high load on the network resources or because it would cause damage to other users.

2.2.4 Agents and Objects

Object-oriented programmers often fail to see any thing novel or new in the idea of agents. When one stops to consider the relative properties of agents and objects, this is perhaps not surprising. Objects are defined as computational entities that *encapsulate* some state, are able to perform actions, or *methods* on this state, and communicate by message passing. While there are obvious similarities, there are also significant differences between agents and objects. The first is in the degree to which agents and objects are autonomous. Recall that the defining characteristic of object-oriented programming is the principle of encapsulation the idea that objects can have control over their own internal state. In programming languages like `JAVA`, we can declare instance variables(and methods) to be `private`, meaning they are only accessible from within the object. (We can of course also declare them `public`, meaning that they can be accessed from anywhere, and indeed we must do this for methods so that they can be used by other objects. But the use of public instance variables is usually considered poor programming style.) In this way, an object can be thought of as exhibiting autonomy over its state: it has control over it. But an object does not exhibit control over it's *behaviour*. That is, if a method m is made available for other objects to invoke, then they can do so whenever they wish once an object has made a method `public`, then it subsequently has no control over whether or not that method is executed. Of course, an object *must* make methods available to other objects, or else we would be unable to build a system out of them. This is not normally an issue, because if we build a system, then we design the objects that go in it, and they can thus be assumed to share a "common goal". But in many types of multi-agent system, (in particular, those that contain agents built by different organisations or individuals), no such common goal can be assumed. It cannot be for granted that an agent i will execute an action (method) a just because another agent j wants it to a may not be in the best interests of i. We thus do not think of agents as invoking methods upon one-another, but rather as *requesting* actions to be performed. If j requests i to perform a, then i may perform the action or it may not. The locus of control with respect to the decision about whether to execute an action is thus different in agent and object systems. In the object-oriented case, the decision lies with the object that invokes the method. In the agent case, the decision lies with the agent that receives the request. The distinction between objects and agents nicely summarised in the following slogan:

Objects do it for free; agents do it for money.

Note that there is nothing to stop us implementing agents using object-oriented techniques. For example, we can build some kind of decision making about whether to execute a method into the method itself, and in this way

achieve a stronger kind of autonomy for our objects. The point is that autonomy of this kind is not a component of the basic object-oriented model.

The second important distinction between object and agent systems is with respect to the notion of *flexible* (reactive, pro-active, social) autonomous behaviour. The standard object model has nothing to say about how to build systems that integrate these types of behaviour. Again, one could object that we can build object-oriented programs that *do* integrate these types of behaviour. But this argument misses the point, which is that the standard object-oriented programming model has nothing to do with these types of behaviour.

The third important distinction between the standard object model and agent systems is that agents are each considered to have their own thread of control in the standard object model, there is a single thread of control in the system. Of course, a lot of work has recently been devoted to *concurrency* in object-oriented programming. For example, the `JAVA` language provides built-in constructs for multi-threaded programming. There are also many programming languages available (most of them admittedly prototypes) that were specifically designed to allow concurrent object-based programming. But such languages do not capture the idea of agents as *autonomous* entities. Perhaps the closest that the object-oriented community comes is in the idea of *active objects*: An active object is one that encompasses its own thread of control. Active objects are generally autonomous, meaning that they can exhibit some behaviour without being operated upon by another object. Passive objects, on the other hand, can only undergo a state change when explicitly acted upon [Boo94]. However, active objects do not need to exhibit flexible behaviour, differently from agents.

To summarise, objects and agents have at least three distinctions:

- agents are *autonomous*: they embody stronger notion of autonomy than objects, and in particular, they have control on their behaviour, and decide for themselves whether or not to perform an action on request from another agent;

- agents are *smart*: they are capable of flexible (reactive, pro-active, social) behaviour, and the standard object model has nothing to say about such types of behaviour;

- agents are *active*: a multi-agent system is inherently multi-threaded, in that each agent is assumed to have at least one thread of active control.

2.2.5 Agents and Expert Systems

Expert systems were the most important AI technology of the 1980s (see, for example [HRWL83]). An expert system is one that is capable of solving

problems or giving advice in some *knowledge-rich* domain [Jac86]. A classic example of an expert system is MYCIN, which was intended to assist physicians in the treatment of blood infections in humans. MYCIN worked by a process of interacting with a user in order to present the system with a number of (symbolically represented) facts, which the system then used to derive some conclusion. MYCIN acted very much as a *consultant*: it did not operate directly on humans, or indeed any other environment. Thus perhaps the most important distinction between agents and expert systems is that expert systems like MYCIN are inherently *disembodied*. By this, we mean that they do not interact directly with any environment: they get their information not via sensors, but through a user acting as middle man. In the same way, they do not act on any environment, but rather give feedback or advice to a third party. In addition, we do not generally require expert systems to be capable of co-operating with other agents. Despite these differences, some expert systems, (particularly those that perform real-time control tasks), look very much like agents. A good example is the ARCHON system (see [JhMC$^+$96]).

2.3 Agent Architectures

An agent architecture is essentially a map of the internals of an agent – its data structures, the operations that may be performed on these data structures, and the control flow between these data structures. Generally speaking, there are two different approaches to the design of agent systems: *micro-architecture* and *macro-architecture*. The former focuses on the characteristics of an agent as stand-alone entity, whereas the latter is related to agents acting as a group, i.e. the social level.

This section discusses a number of different types of agent architecture, first by an abstract point of view by surveying some fairly high-level decisions, and then from a more concrete point of view - distinguishing between the micro and macro aspects - on the data structures and algorithms that can be present within an agent or a society of agents.

2.3.1 Abstract Architectures

To introduce a simple formal model of agents, let us assume that the the state of the agent's environment can be characterised as a set S of *environment states*

$$S = \{s_1, s_2, \ldots s_n\}.$$

We can consider S as a set of discrete, instantaneous states, since it is a fairly standard assumption that any continuous environment can be modelled by a discrete environment to any desidered degree of accuracy. Agents are assumed to have a repertoire of possible *actions* available to them, represented

by the finite set A:

$$A = \{a1, a2, \ldots\}.$$

Thus, an agent can be abstractly viewed as a function

$$action : S^* \rightarrow A$$

which maps sequences of environment states to actions. S^* is the set of sequences of elements of S. Intuitively, an agent decides what action to perform on the basis of its history, i.e. its experiences to date. These experiences are represented as a sequence of environment states, i.e. those that the agent has encountered thus far. The behaviour of a non-deterministic environment can be modelled as a function (being $\wp(S)$ the power set of S):

$$env : S \times A \rightarrow \wp(S)$$

which takes the current state of the environment $s \in S$ and an action $a \in A$ (performed by the agent), and maps them to a set of environment states $env(s, a)$ those that could result from performing action a in state s. Note that if all the sets in the range of env are all singletons, (i.e., if the result of performing any action in any state is a set containing a single member), then the environment is deterministic, and its behaviour can be accurately predicted.

We can represent the interaction of agent and environment as a *history* (or *run*). A history h is a sequence:

$$h : s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \ldots$$

where s_0 is the initial state of the environment (i.e., its state when the agent starts executing), a_u is the u'th action that the agent chose to perform, and s_u is the u'th environment state (which is one of the possible results of executing action a_{u-1} in state s_{u-1}). We will denote by $hist(agent, environment)$ the set of all *histories* of *agent* in *environment*.

Two agents ag_1 and ag_2 are said to be *behaviourally equivalent* with respect to environment env if and only if $hist(ag_1, env) = hist(ag_2, env)$, and simply behaviourally equivalent if and only if they are behaviourally equivalent with respect to all environments. In general, we are interested in agents whose interaction with their environment does not end, i.e., they are *non-terminating*. In such cases, the histories that we consider will be infinite.

Purely Reactive Agents

Some agents decide what to do without reference to their history. In other words, they base their decision making entirely on the present, with no reference at all to the past. Such agents are called *purely reactive*, since

they simply respond directly to their environment. Formally, the behaviour of a purely reactive agent can be represented by a function

$$action : S \rightarrow A$$

For each purely reactive agent, there is an equivalent standard agent; the reverse is not generally the case. A thermostat is an example of a purely reactive agent. Assume, without loss of generality, that the thermostat's environment can be in one of two states either too cold, or temperature OK. Then the thermostat's action function is simply:

$$action(S) = \begin{cases} \text{heater off} & \text{if } e = \text{temperature OK}, \\ \text{heater on} & \text{otherwise.} \end{cases}$$

Perception

Viewing agents at this abstract level does not help to construct them, since it gives us no clues about out to design the decision function *action*. Thus, we begin refining the abstract model of agents by breaking it down into two sub-systems: perception and action (see Figure 2.2). The idea is that the

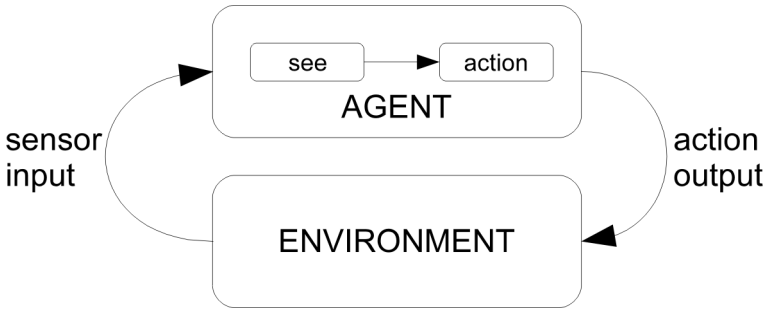


Figure 2.2: Agent's perception and action subsystems.

function *see* captures the agent's ability to observe its environment, whereas the *action* function represents the agent's decision making process. The *see* function might be implemented in hardware in the case of an agent situated in the physical world: for example, it might be a video camera or an infra-red sensor on a mobile robot. For a software agent, the sensors might be system commands that obtain information about the software environment. The *output* of the *see* function is a *percept* a perceptual input. Let P be a (non-empty) set of percepts. Then *see* is a function

$$see : E \rightarrow P$$

which maps environment states to percepts, and *action* is a function

$$action : P^* \rightarrow A$$

which maps sequences of percepts to actions. An agent ag is then considered to be a pair $Ag =< see, action >$, consisting of a *see* function and an *action* function.

Agents with state

Modelling an agent as a decision function from sequences of percepts to actions, can be a somewhat unintuitive representation. An equivalent but more natural scheme considers agents that maintain state (see Figure 2.3) These agents have some internal data structure, which is typically used to

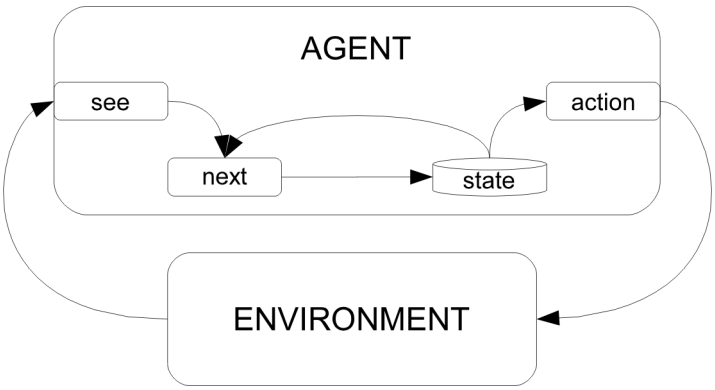


Figure 2.3: Agents that mantain state.

record information about the environment state and history. Let I be the set of all internal states of the agent. The perception function *see* for a state-based agent is unchanged:

$$see : S \rightarrow P$$

The action-selection function *action* is now defined as a mapping

$$action : I \rightarrow A$$

from internal states to actions. An additional function *next* is introduced, which maps an internal state and percept to an internal state:

$$next : I \times P \rightarrow I.$$

The behaviour of a state-based agent can be summarized as follows. The agent starts in some initial internal state i_0. It then observes its environment state s, and generates a percept $see(s)$. The internal state of the agent is then updated via the *next* function, becoming set to $next(i_0, see(s))$. The action selected by the agent is then $action(next(i_0, see(s)))$. This action is then performed, and the agent enters another cycle, perceiving the world

via *see*, updating its state via *next*, and choosing an action to perform via *action*. It is worth observing that state-based agents as defined here are in fact no more powerful than the standard agents introduced earlier. In fact, they are identical in their expressive power, since every state-based agent can be transformed into a standard agent that is behaviourally equivalent.

2.3.2 Micro Architectures

So far, we have considered agents only in the abstract, without discussing how to implement their functionality. In this section we move away from the abstract view of agents, and begin to make quite specific commitments about the internal structure and operation of agents. In each subsection, the nature of these commitments, as well as the assumptions upon which the architectures depend, and the relative advantages and disadvantages of each, are briefly explained. In particular, we will consider four classes of agents architectures:

- **symbolic/logic-based architectures:** in which decision making is realised through logical deduction; originally (1956-1985), pretty much all agents designed within AI were symbolic reasoning agents. Its purest expression proposes that agents use explicit logical reasoning in order to decide what to do.
- **reactive architectures:** in which decision making is implemented in some form of direct mapping from situation to action; problems with symbolic reasoning led to a reaction against this – the so-called reactive agents movement, 1985present.
- **hybrid architectures:** which attempt to combine the best of reasoning and reactive architectures; they have been proposed since 1990; a significant case is represented by layered architectures – in which decision making is realised via various software layers, each of which is more-or-less explicitly reasoning about the environment at different levels of abstraction;
- **belief-desire-intention architectures:** in which decision making depends upon the manipulation of data structures representing the beliefs, desires, and intentions of the agent.

Logic-based

Logic-based architectures are the oldest ones in building agents. According to them, the decision making strategy of an agent is viewed as *deduction*: agent's program is encoded as a logical theory, and the process of selecting an action is reduced to a problem of proof. Such architectures have the advantage of semantic clarity, and in addition allow us to bring to bear all the

apparatus of logic and theorem proving that has been developed in AI and computer science over the years. In other words, logic-based architectures are elegant, and have a clean logical semantics – wherein lies much of their long-lived appeal. However, such architectures have many disadvantages. In particular, the inherent computational complexity of theorem proving makes it questionable whether agents as theorem provers can operate effectively in time-constrained environments. Decision making in such agents is predicated on the assumption of calculative rationality the assumption that the world will not change in any significant way while the agent is deciding what to do, and that an action which is rational when decision making begins will be rational when it concludes. The problems associated with representing and reasoning about complex, dynamic, possibly physical environments are also essentially unsolved.

Reactive

Reactive architectures eschew representations and models in favour of a closer relationship between agent perception and action. They are more economical in computational terms, and well-suited to episodic environments that require real-time performance. However, the process of engineering such architectures in not well understood.

The seemingly intractable problems with symbolic/logical approaches for building agents led some researchers to question, and ultimately reject, the assumptions upon which such approaches are based. These researchers have argued that minor changes to the symbolic approach, such as weakening the logical representation language, will not be sufficient to build agents that can operate in time-constrained environments: nothing less than a whole new approach is required. In the mid-to-late 1980s, these researchers began to investigate alternatives to the symbolic AI paradigm. It is difficult to neatly characterise these different approaches, since their advocates are united mainly by a rejection of symbolic AI, rather than by a common manifesto. Alternative approaches to agency are sometime referred to as *behavioural* (since a common theme is that of developing and combining individual behaviours), *situated* (since a common theme is that of agents actually situated in some environment, rather than being disembodied from it), and finally *reactive* (because such systems are often perceived as simply reacting to an environment, without reasoning about it).

The *subsumption architecture* is arguably the best-known reactive agent architecture [Bro85]. It was developed by Rodney Brooks, one of the most vocal and influential critics of the symbolic approach to agency that have emerged in the literature. There are two defining characteristics of the subsumption architecture.

The first is that an agent's decision-making is realised through a set of *task accomplishing behaviours*; each behaviour may be though of as an

individual *action* function, which continually takes perceptual input and maps it to an action to perform. Each of these behaviour modules is intended to achieve some particular task. In Brooks' implementation, the behaviour modules are finite state machines. An important point to note is that these task accomplishing modules are assumed to include *no* complex symbolic representations, and are assumed to do *no* symbolic reasoning at all. In many implementations, these behaviours are implemented as rules of the form *situation* $\rightarrow$ *action* which simple map perceptual input directly to actions.

The second defining characteristic of the subsumption architecture is that many behaviours can "fire" simultaneously. There must obviously be a mechanism to choose between the differentactions selected by these multiple actions. Brooks proposed arranging the modules into a subsumption hierarchy, with the behaviours arranged into layers (see Figure 2.4). Lower layers in the hierarchy are able to inhibit higher layers: the lower a layer is, the higher is its priority. The idea is that higher layers represent more abstract behaviours.

Figure 2.4: The subsumption architecture.

In short, there are obvious advantages to reactive approaches such as that Brooks' subsumption architecture: simplicity, economy, computational tractability, robustness against failure, and elegance all make such architectures appealing. But there are some fundamental, unsolved problems, not just with the subsumption architecture, but with other purely reactive architectures:

- If agents do not employ models of their environment, then they must have sufficient information available in their *local* environment for them to determine an acceptable action.
- Since purely reactive agents make decisions based on *local* information, (i.e., information about the agents *current* state), it is difficult

to see how such decision making could take into account *non-local* information it must inherently take a "short term" view.

- It is difficult to see how purely reactive agents can be designed that *learn* from experience, and improve their performance over time.
- A major selling point of purely reactive systems is that overall behaviour *emerges* from the interaction of the component behaviours when the agent is placed in its environment. But the very term "emerges" suggests that the relationship between individual behaviours, environment, and overall behaviour is not understandable. This necessarily makes it very hard to *engineer* agents to fulfill specific tasks. Ultimately, there is no principled *methodology* for building such agents: one must use a laborious process of experimentation, trial, and error to engineer an agent.
- While effective agents can be generated with small numbers of behaviours (typically less that ten layers), it is *much* harder to build agents that contain many layers. The dynamics of the interactions between the different behaviours become too complex to understand.

Hybrid

Given the requirement that an agent be capable of reactive and pro-active behaviour, an obvious decomposition involves creating separate subsystems to deal with these different types of behaviours. This idea leads naturally to a class of architectures (i.e. layered architectures) in which the various subsystems are arranged into a hierarchy of interacting layers. In layered agent architectures, decision making is partitioned into a number of different decision making layers, each dealing with the agent's environment at a different level of abstraction; they provide a natural way of decomposing agent functionality, and are currently a popular approach to agent design.

Typically, there will be at least two layers, to deal with reactive and pro-active behaviours respectively. In principle, there is no reason why there should not be many more layers. However, many layers there are, a useful typology for such architectures is by the information and control flows within them. Broadly speaking, we can identify two types of control flow within layered architectures: horizontal layering and vertical layering (see Figure 2.5 and Figure 2.6, respectively):

Horizontal layering. In horizontally layered architectures (Figure 2.5), the software layers are each directly connected to the sensory input and action output. In effect, each layer itself acts like an agent, producing suggestions as to what action to perform.

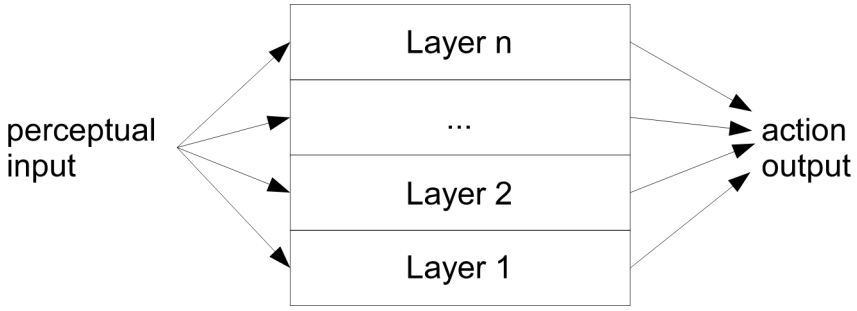


Figure 2.5: Horizontal Layered Architectures.

The great advantage of horizontally layered architectures is their conceptual simplicity: if we need an agent to exhibit n different types of behaviour, then we implement n different layers. However, because the layers are each in effect competing with one-another to generate action suggestions, there is a danger that the *overall* behaviour of the agent will not be coherent. In order to ensure that horizontally layered architectures *are* consistent, they generally include a *mediator* function, which makes decisions about which layer has "control" of the agent at any given time. The need for such central control is problematic: it means that the designer must potentially consider all possible interactions between layers. If there are n layers in the architecture, and each layer is capable of suggesting m possible actions, then this means there are m^n such interactions to be considered. This is clearly difficult from a design point of view in any but the most simple system. The introduction of a central control system also introduces a *bottleneck* into the agent's decision making. These problems are partly alleviated in a vertically layered architecture.

Vertical layering. In vertically layered architectures (Figure 2.6a and 2.6b), sensory input and action output are each dealt with by at most one layer each.

Vertically layered architectures can be subdivided into *one pass* architectures (Figure 2.6a) and *two pass* architectures (Figure 2.6b). In one-pass architectures, control flows sequentially through each layer, until the final layer generates action output. In two-pass architectures, information flows up the architecture (the first pass) and control then flows back down. There are some interesting similarities between the idea of two-pass vertically layered architectures and the way that organisations work, with information flowing up to the highest levels of the organisation, and commands then flowing down. In both one pass and two pass vertically layered architectures, the complexity of interactions between layers is reduced: since there are $n - 1$ interfaces between n layers, then if each layer is capable of suggesting m actions, there are at most $m^2(n-1)$ interactions to be considered

between layers. This is clearly much simpler than the horizontally layered case. However, this simplicity comes at the cost of some flexibility: in order for a vertically layered architecture to make a decision, control must pass between each different layer. This is not fault tolerant: failures in any one layer are likely to have serious consequences for agent performance.

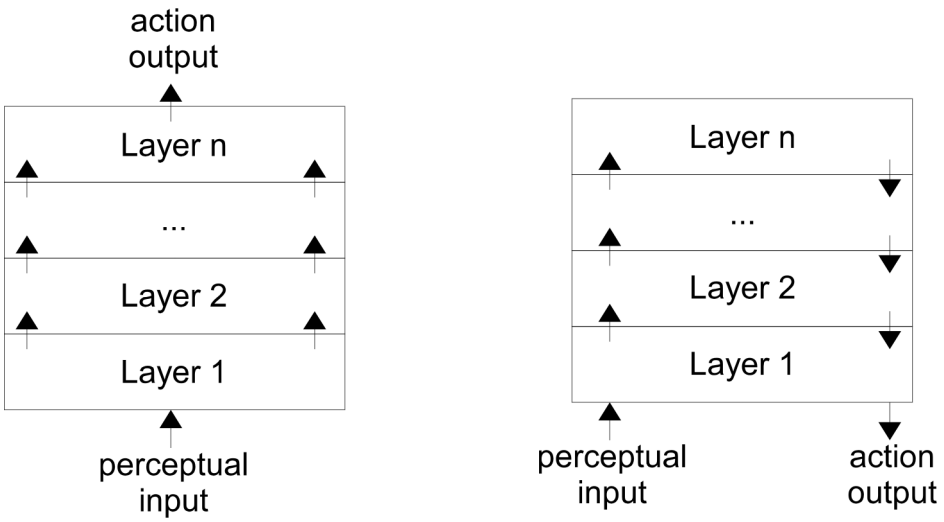


Figure 2.6: Vertical Layered Architectures: (a) one pass control; (b) two pass control.

To summarize, layered architectures are currently the most popular general class of agent architecture available. Layering represents a natural decomposition of functionality: it is easy to see how reactive, pro-active, social behaviour can be generated by the reactive, pro-active, and social layers in an architecture. The main problem with layered architectures is that while they are arguably a pragmatic solution, they lack the conceptual and semantic clarity of unlayered approaches. In particular, while logic-based approaches have a clear logical semantics, it is difficult to see how such a semantics could be devised for a layered architecture. Another issue is that of interactions between layers. If each layer is an independent activity producing process (as in TOURINGMACHINES [Fer92]), then it is necessary to consider all possible ways that the layers can interact with one another. This problem is partly alleviated in two-pass vertically layered architecture such as INTERRAP [Mül96].

BDI

Belief-desire-intention (BDI) architectures have their roots in the theory of human *practical reasoning* developed by the philosopher Michael Bratman [Bra87] in the mid 1980s. Pratical reasoning is the process of deciding,

moment by moment, which action to perform in the furtherance of our goals. The conceptual framework of the BDI model is described in [BIP91].

Practical reasoning involves two important processes: deciding *what* goals we want to achieve, and *how* we are going to achieve these goals. The former process is known as *deliberation*, the latter as *means-ends* reasoning. Intentions play a number of important roles in practical reasoning:

- Intentions drive means-ends reasoning;
- Intentions constrain future deliberation;
- Intentions persist;
- Intentions influence beliefs upon which future practical reasoning is based.

A key problem in the design of practical reasoning agents is that of of achieving a good *balance* between these different concerns.

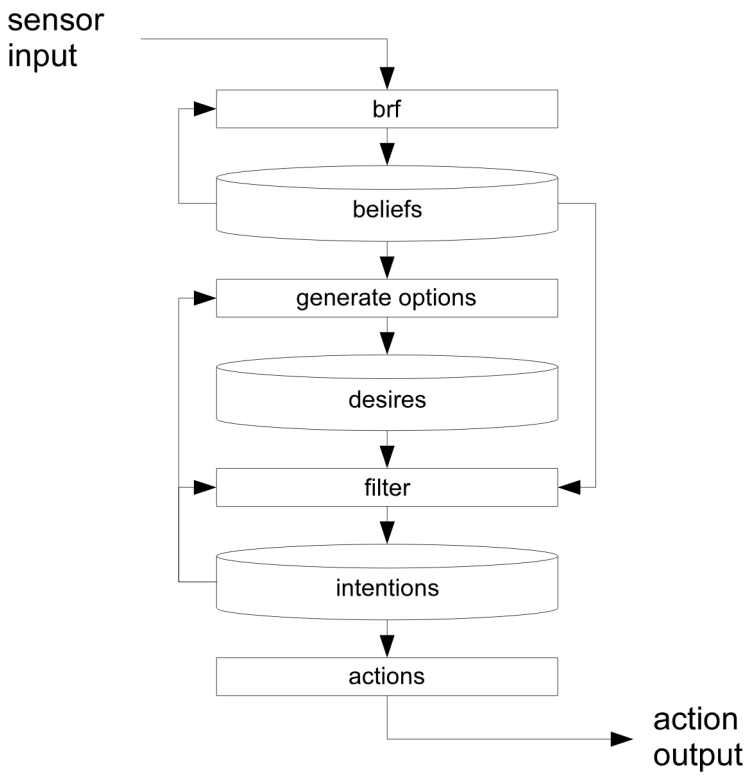


Figure 2.7: The Belief-Desire-Intention architecture.

The process of practical reasoning in a BDI agent is summarised in Figure 2.7. As this Figure illustrates, there are seven main components to a BDI agent:

- a set of current *beliefs*, representing information the agent has about its current environment;
- a *belief revisionfunction (brf)*, which takes a perceptual input and the agent's current beliefs, and on the basis of these, determines a new set of beliefs;
- an *option generation function (options)*, which determines the options available to the agent (its desires), on the basis of its current beliefs about its environment and its current *intentions*;
- a set of current options, representing possible courses of actions available to the agent;
- a *filter* function *(filter)*, which represents the agent's deliberation process, and which determines the agent's intentions on the basis of its current beliefs, desires, and intentions;
- a set of current *intentions*, representing the agent's current focus those states of affairs that it has committed to trying to bring about;
- an *action selection* function *(execute)*, which determines an action to perform on the basis of current intentions.

In short, BDI architectures are practical reasoning architectures, in which the process of deciding what to do resembles the kind of practical reasoning that we appear to use in our everyday lives. The basic components of a BDI architecture are data structures representing the beliefs, desires, and intentions of the agent, and functions that represent its deliberation (deciding *what* intentions to have i.e., decidingwhat to do) and means-ends reasoning (deciding how to do it). Intentions play a central role in the BDI model: they provide stability for decision making, and act to focus the agent's practical reasoning. A major issue in BDI architectures is the problem of striking a *balance* between being committed to and overcommitted to one's intentions: the deliberation process must be finely tuned to its environment, ensuring that in more dynamic, highly unpredictable domains, it reconsiders its intentions relatively frequently in more static environments, less frequent reconsideration is necessary.

The BDI model is attractive for several reasons. First, it is intuitive we all recognise the processes of deciding what to do and then how to do it, and we all have an informal understanding of the notions of belief, desire, and intention. Second, it gives us a clear functional decomposition, which indicates what sorts of subsystems might be required to build an agent. But the main difficulty, as ever, is knowing how to efficiently implement these functions.

2.3.3 Macro-Architectures

The environment in which agents operate might contain or not other agents. Although there are situations where an agent can operate usefully by itself, the increasing interconnection of computers is making such situations rare, and usually an agent interacts with other agents. To cope with environments that contain societies of agents, a computational infrastructure that includes protocols for agents to communicate and interact, is needed. In particular, the environments must provide a computational structure for interactions that includes communication protocols and interation protocols. Communication protocols allow agents to exchange and understand messages; interaction protocols enable agents to have conversations (structured exchange of messages).

A Multi-Agent System (MAS) is a system composed of a population of autonomous agents, which cooperate with each other to reach common objectives, while simultaneously each agent pursues individual objectives. The rationale for interconnetting agents is to enable them to cooperate in solving problems, to share expertise, to work in parallel on common problems, to be developed and implemented modularly, to be fault tolerant through redudancy, to represent multiple viewpoints and the knowledge of multiple experts, and to be reusable.

Multiagent systems are a relatively new field of computer science. They have only been studied since about 1980, and they gained widespread recognition since about the mid-1990s. International interest in the field has then grown enormously, since agents are considered a suitable software paradigm to exploit the possibilities presented by massive open distributed systems such as the Internet.

There is a popular slogan in the multiagent systems community:

"There's no such thing as a single agent system."

The point of the slogan is that interacting systems are the norm in the everyday computing world. Almost all the systems contain a number of sub-systems that must interact each other to successfully carry out their tasks. The typical multiagent system contains a number of agents, which interact with one another through communication.

The agents are able to act in an environment; different agents have different "spheres of influence", in the sense that they will have control over - or at least be able to influence - different parts of the environment. These spheres of influence may coincide in some cases, thus giving rise to dependency relationships among the agents. Agents will also typically be linked by other relationships, such as "power" relationships, where one agent fully controls another agent. When facing with multiagent domains, it is critical to understand the *type* of interaction that take place between the agents.

MultiAgent Systems and Distributed Artificial Intelligence

Multiagent systems are strictly related to the context of Distributed Artificial Intelligence (DAI). The broad scope and the multi-disciplinary nature of the Distributed Artificial Intelligence make it difficult to characterize DAI in a few words. As starting point, it can be said that DAI is the study, construction, and application of multi-agent systems. Traditionally, two main types of DAI systems have been distinguished: multiagent systems and distributed problem-solving systems. In the first ones, several agents coordinate their knowledge, activity and reason about the processes of coordination. In the second ones, the work of solving a particular problem is divided among a number of nodes that divide and share knowledge about the problem and the developing solution. The modern concept of multiagent systems covers both type of systems. The role that the concept of a multiagent system plays in DAI is comparable to the role the the concept of an individual agent plays in traditional AI. The elementary question faced by DAI is "When and how should which agents interact – cooperate and compete – to successfully meet their design objectives?"

An agent has the ability to *communicate*. This ability is part perception (the receiving of messages) and part action (the sending of messages). Communication can enable the agents to *coordinate* their actions and behavior.

Coordination is a property of system of agents performing some activity in a shared environment.

Cooperation is coordination among non-antagonistic agents, while negotiation is coordination among competitive or simply self-interested agents. A problem of a multiagent system is how it can mantain global coherence without explicit global control. In this case, the agents must be able on their own to determine goals they share with other agents.

Communication. In order for a MAS to solve common problems coherently, the agents must communicate amongst themselves, coordinate their activities. Coordination and communication are central to MAS, for without it, any benefits of interaction vanish and the group of agents quickly degenerates into a collection of individuals with a chaotic behaviour.

Coordination. There are several reasons why multiple agents need to be coordinated:- Prevent chaos. No agent possesses a global view of the entire agency to which it belongs, as this is simply not feasible in any community of reasonable complexity. Consequently, agents have only local views, goals and knowledge that may interfere with rather than support other agents' actions. Coordination is vital to prevent chaos during conflicts. Meet global constraints. Agents performing network management may have to respond to certain failures within seconds and others within hours. Coordinating agents' behaviour is therefore essential to meet such a goal. - Agents in

MAS possess different capabilities and expertise. Therefore, agents need to be coordinated in just the same way that different medical specialists, including anesthetists, surgeons, ambulance personnel, nurses, etc., need to coordinate their capabilities to treat a patient. - Agent's actions are frequently interdependent and hence an agent may need to wait for another agent to complete its task before executing its own. Such interdependent activities need to be coordinated.

The most renowned coordination is based on the Contract-Net Protocol (CNP). In this approach, a decentralized market structure is assumed and agents can take on two roles: a manager and a contractor. The basic premise of this form of coordination is that if an agent cannot solve an assigned problem using local resources/expertise, it will decompose the problem into sub-problems and try to find other willing agents with the necessary resources/expertise to solve these sub-problems. Assigning the sub-problems is solved by a contracting mechanism. It consists of contract announcement by the manager agent, submission of bids by contracting agents in response to the announcement, and the evaluation of the submitted bids by the manager, which leads to awarding a sub-problem contract to the contractor(s) with the most appropriate bid(s). Another important component of agent scheduling is the communication protocols among agents. In order to achieve this coordination, the agents might have to interact and exchange information; therefore they need to communicate by sending messages. KQML (Knowledge Query and Manipulation Language) is a good example of communication language.

2.4 Agent Languages

The need for software support tools for the design and implementation of agent systems was identified as long ago as the mid-1980s (see, for example [GBH88]).

2.4.1 Agent-oriented programming

The *agent-oriented programming* is a new programming paradigm, based on a social view of computation, which has been proposed by Yoav Shoham [Sho93]. The main idea is to directly program agents in term of *mentalistic* notions, such as belief, desire, and intention, which represent the properties of agents. The motivation behind the proposal is that humans use such concepts as an *abstraction* mechanism for representing the properties of complex systems: it might be useful to use them to program machines, in the same way that we use these notions to describe the behaviour of humans.

AGENT0

The first implementation of the agent-oriented programming paradigm was the AGENT0 programming language. In this language, an agent is specified in terms of a set of capabilities (things that the agent can do), a set of initial *beliefs* (playing the role of beliefs in BDI architectures), a set of initial commitments (playing a role similar to that of intentions in BDI architectures), and a set of *commitment rules*. The key component, which determines how the agent acts, is the commitment ruleset. Each commitment rule contains a *message condition*, a *mental condition*, and an action. In order to determine whether such a rule fires, the message condition is matched against the messages the agent has received; the mental condition is matched against the beliefs of the agent. If the rule fires, then the agent becomes committed to the action. Actions may be *private*, corresponding to an internally executed subroutine, or *communicative*, i.e., sending messages. Messages are constrained to be one of three types: "requests" or "unrequests" to perform or refrain from actions, and "inform" messages, which pass on information. Request and unrequest messages typically result in the agent's commitments being modified; inform messages result in a change to the agent's beliefs.

It is important to note that AGENT0 language is essentially a prototype, not intended for building anything like large-scale production systems. The main drawback is that the relationship between the logic and interpreted programming language is only loosely defined. The programming language cannot be said to truly *execute* the associated logic.

Concurrent METATEM

Fisher made a stronger claim in the respect of the association between language and logic by introducing of the Concurrent METATEM language [Fis94], which is based on the direct execution of logical formulae. A Concurrent METATEM system contains a number of concurrently executing agents, each of them being able to communicate with its peers via asynchronous broadcast message passing. Each agent is programmed by giving it a *temporal logic* specification of the behaviour that it is intended the agent should exhibit. An agent's specification is executed directly to generate its behaviour. Execution of the agent program corresponds to iteratively building a logical model for the temporal agent specification. It is possible to prove that the procedure used to execute an agent specification is correct, in that if it is possible to satisfy the specification, then the agent will do so.

The logical semantics of Concurrent METATEM are closely related to the semantics of temporal logic itself. This means that, amongst other things, the specification and verification of Concurrent METATEM systems is a realistic proposition.

An agent program in Concurrent METATEM has the form $\Lambda_i P_i \Rightarrow F_i$,

where P_i is a temporal logic formula referring only to the present or past, and F_i is a temporal logic formula referring to the present or future. The $P_i \Rightarrow F_i$ formulae are known as *rules*. The basic idea for executing such a program may be summed up in the following slogan:

on the basis of the past, *do* the future.

Thus, each rule is continually matched against an internal, recorded *history*, and if a match is found, then the rule *fires*. If a rule fires, then any variables in the future time part are instantiated, and the future time part then becomes a *commitment* that the agent will subsequently attempt to satisfy. Satisfying a commitment typically means making some predicate true within the agent.

Concurrent METATEM is a good illustration of how a quite pure approach to logic-based agent programming can work, even with a quite expressive logic.

Related Work

Since Shoham's proposal, a number of agent-oriented languages have been proposed. Some examples include Becky Thomas's Planning Communicating Agents (PLACA) language [Tho93], MAIL [HSM94], and Anand Rao's AGENTSPEAK(L) language [Rao96]. APRIL is a language that is intended to be used for building multi-agent systems, although it is not "agent-oriented" in the sense that Shoham describes. The TELESCRIPT programming language, developed by General Magic, Inc., was the first mobile agent programming language [Whi]. That is, it explicitly supports the idea of agents as processes that have the ability to autonomously move themselves across a computer network and recommence executing at a remote site.

Since TELESCRIPT was announced, a number of mobile agent extensions to the Java programming language have been developed. One of the most important extension is JADE (Java Agent DEvelopment Framework) [BPR00], a software framework fully implemented in Java language. It supports the implementation of multi-agent systems through a middle-ware that complies with the FIPA specifications. The infrastructure of the proposed multiagent architecture presented in chapterch:pacmas is based on JADE.

Chapter 3

Planning

The previous chapter introduced the most important capabilities that an agent should own, to exhibit a flexible behaviour. This chapter focuses on the aspects regarding the *proactive* behavior, i.e. its "goal-oriented" behaviour. In fact, to achieve their goals, agents often need to act in the world. Acting in complex environments requires the ability of considering which actions should be taken in order to achive a goal, that is what AI planning systems attempt to do. In other words, agent's proactive behaviour is strictly related to its *planning* capabilities.

People, the best exemplars of intelligence we have to date, do a lot of planning. It should be not surprising that investigating the planning process has always been a central part of research in the field of Artificial Intelligence. In particular, the quest of building intelligent agents has forced researchers to investigate algorithms for generating appropriate actions in a timely fashion.

This chapter aims to introduce the reader to the main concepts related to planning. Section 3.2 begins by illustrating the simplifying assumptions that characterize classical planning systems, defines formally the main concepts regarding planning context, reviews the languages used to represent planning problems, and describes the main approaches employed to solve planning problems. Section 3.3 sketches the main issues related to planning in more complex domains, such as those encountered in enviroments that model real worlds. Section 3.4 presents planning by abstraction, one of the most effective technique to manage complexity in planning systems. Finally, section 3.5 surveys the most important planning systems that have given a contributions to the development of automated planning in general.

3.1 Introduction

Since the mid 1960's, researchers (mostly from the artificial intelligence community) have studied how computers can plan, including how to classify

planning problems, how to represent them formally, how to solve them, and how to integrate the resulting systems into human-machine environments.

Planning involves finding a sequence of actions that solve some problem in a specific domain. Given a domain definition and a problem, a planner is asked to find a solution to the problem.

A *planning domain* is defined by a set of legal operators and states.

The legal operators are defined in terms of *preconditions* and *effects*, where the preconditions must be satisfied before the operator can be applied, and the effects describe the changes to the state in which the operator is applied.

States are snapshots of the world that include all of its aspects that are relevant for planning. They are composed of a set of conditions that describe the relevant features of a model of the world.

A *planning problem* consists of an initial state (*init*), which describes the initial configuration of the world, and a set of goal conditions (*goal*) to be achieved, which describes the desired configuration.

A solution to a problem is called *plan*, which consists of a sequence of operators that transform the given initial state into some final state that satisfies the goal.

Thus, a simple abstract formulation of the planning problem defines three inputs:

- a description of initial state of the world in some formal language;
- a description of the agent's goal (i.e., what behaviour is desired) in some formal language;
- a description of the possible actions that can be performed (i.e., the domain theory).

The planner's output is a sequence of actions which, when executed in any world satisfying the initial state description, will achieve the goal.

3.2 Classical planning

There are many possibilities to represent the world, agent's goal, and possibile actions. However, the task of writing a planning algorithm is harder for more expressive representation languages, and the speed of the resulting algorithm descreases accordingly. In general, classical planners make the following simplifying assumptions:

- **Atomic Actions:** The execution of an action is indivisible and uninterruptible, so we need not to consider the state of the world during the execution process. Instead, execution can be modeled as an atomic trasformation from one world state to another. Simultaneously executed actions are impossible;

- **Deterministic Effects:** The effect of executing any action is a deterministic function of the action and the state of the world when the action is executed;
- **Omniscience:** The agent has complete knowledge of the initial state of the world and of the nature of its own actions;
- **Static Environment:** The only way the world changes is by the agent's own actions. There are no other agents and the world is static. This assumption means that the world description needs to be specified only by the initial state of the world.

In short, classical planning assumes that actions and the environment are always completely predictable and observable. Despite these assumptions are unrealistic, they make the planning problem more tractable. However, they often make hard to face the problem of planning in real environments. Thus, a class of planning algorithms that relaxes one or more of the above assumptions has been devised, which will be sketched in section 3.3.

3.2.1 Formal Definitions

A problem space Σ is formally defined by a triple (L, S, O), where L is a first-order language, S is a set of states, and O is a set of operators. Each state $S_i \in S$ is a finite and consistent set of atomic sentences in L.

Each operator $\alpha \in O$ is defined by a triple $(P_\alpha, D_\alpha, A_\alpha)$, where P_α, the preconditions, are a set of literals (i.e. positive or negative atomic sentences) in L, and both the deletes D_α and adds A_α are finite sets of atomic sentences in L. The combination of the adds and the deletes comprise the effects of an operator E_α, such that if $p \in A_\alpha$ then $p \in E_\alpha$ and if $p \in D_\alpha$ then $(\neg p) \in E_\alpha$.

A problem ρ consists of two components:

- An initial state $S_0 \in S$, where S_0 is a description of an initial state of the world;
- A goal state $S_g \in S$, where S_g is a partial description of a desired state.

The solution (or plan) Π to a problem is a sequence of operators that transforms the initial state S_0 into some final state S_n that satisfies the goal state S_g. A plan is composed by the concatenation of operators or subplans.

Let $A : O \times S \rightarrow S$ be an application procedure that applies an operator to a state to produce a new state by removing the deleted literals, and inserting the added literals. For any state S_i (where $\setminus$ represents set difference),

$$A(\alpha, S_i) = (S_i \setminus D_\alpha) \cup A_\alpha.$$

The application procedure can be extended to apply to plans in the obvious way, where each operator applies to each of the resulting states in sequence. Thus, given the initial state S_o, a plan $\Pi = \alpha_1; \ldots; \alpha_n$ defines a sequence of states $S_1, \ldots, S_n$, where

$$S_i = A(\alpha_1; \ldots; \alpha_i, S_0) = A(\alpha_i, S_{i-1});\ 1 \leq i \leq n$$

A plan Π is correct whenever the preconditions of each operator are satisfied in the state in which the operator is applied:

$$P_{\alpha_i} \subseteq S_{i-1};\ 1 \leq i \leq n$$

Π solves a problem $\rho = (S_0, S_g)$ whenever Π is correct and the goal S_g is satisfied in the final state: $S_g \subseteq A(\Pi, S_0)$.

3.2.2 Representations of Planning Problems

A representation language allows specifying planning problems in a way that can be understood by a computer. In general, more expressive representation languages allow modeling a greater variety of planning problems but are also more complicated, which makes it harder for humans to encode planning problems and to understand planning problems that have been encoded by others, and also harder for computers to solve encoded planning problems. Researchers originally tried to use logic as representation language and theorem-proving techniques to find valid plans by proving that there exists an action sequence that transforms the initial state to a goal state. However, logic is very expressive and thus theorem proving turned out to be slow. Therefore, specialized representation languages for planning problems have been devised.

STRIPS

One of the earliest languages for describing planning problems was used by a planning system called STRIPS [FN71]. The acronym STRIPS stands for "Stanford Research Institute Problem Solver", a very famous planner built in the early 1970s to control an unstable mobile robot, affectionately known as "Shakey". The STRIPS representation language is still widely used today.

The expressive power of STRIPS is the same of propositional logic: states are specified using a subset of logic, namely as conjunctions of propositions, that is, as sets of statements that are either true or false. STRIPS follows the so called "Closed World Assumption" [Rei80]: all atomic formulae explicitly listed in the state description are assumed to be true, whereas unlisted propositions are assumed to be false.

In the STRIPS representation, actions are compactly represented with action schemata, that is, parameterized descriptions that describe in which

states the actions can be executed and which states result from their execution. An action is obtained from an action schema by supplying objects for its parameters. Actions are represented with preconditions and effects. An action can be executed in all states that contain the propositions on its precondition list. The precondition of each action follows the same restriction as the problem's goal: they are a conjunction of positive literals. An action's effect, on the other hand, is a conjunction that may include both positive and negative literals. When an action is executed, it changes the world description in the following way. All the positive literals in the effect conjunction (the action's add-list) are added into the state description, while all the negative literals (the action's delete-list) are removed. In other words, its execution results in the state obtained from the state before its execution by deleting all propositions on its delete list and adding all propositions on its add list. The add and delete lists address the so called frame problem: how to specify the consequences of action executions efficiently, by not listing those propositions that remain unaffected by action executions.

However, the expressive power of STRIPS language is limited. For example, it cannot express the consumption of continuous resources (such as fuel and time). Representation languages such as the Action Description Language (ADL) and the Problem Domain Description Language (PDDL) have therefore extended STRIPS to make it more expressive.

ADL

Despite making the description of the domains more simple, STRIPS imposed several limitations in the representation of real problems. One of the most important restrictions is that literals (and consequently actions) must be function-free. Pednault [Ped89] defined the Action Description Language or ADL, the most important variant of the STRIPS language that tried to narrow the gap between the expressive power of the propositional logic and the first order logic. Table 3.1 briefly compares the ADL with the basic STRIPS language.

PDDL

The demand for a common planning language that allows researchers to exchange benchmark problems and comparing results landed to the definition of a standard syntax, called the "Planning Domain Definition Language" or PDDL [McD98]. It mostly descended from the language of the partial-order planner UCPOP [PW92], which supports a consistent set of ADL. PDDL includes sublanguages for STRIPS, ADL, and the hierarchical task networks [EHN94]. The most important PDDL extensions are:

- Actions with variable parameters

STRIPS Language	ADL Language
Only positive literals in states: $Poor \wedge Unknown$	Positive and negative literals in states: $\neg Rich \wedge \neg Famous$
Closed World Assumption: Unmentioned literals are false.	Open World Assumption: Unmentioned literals are unknown.
Effect $P \wedge \neg Q$ means add P and delete Q.	Effect $P \wedge \neg Q$ means add P and $\neg Q$ and delete $\neg P$ and Q.
Only ground literals in goals: $Rich \wedge Famous$	Quantified variables in goals: $\exists x : At(P_1, x) \wedge At(P_2, x)$ is the goal of having P_1 and P_2 in the same place.
Goals are conjunctions: $Rich \wedge Famous$	Goals allow conjunction and disjunction: $\neg Poor \wedge (Famous \vee Smart)$
Effects are conjunctions.	Conditional effects allowed: when $P : E$ means E is an effect only if P is satisfied.
No support for equality.	Equality predicate $(x = y)$ is built in.
No support for types.	Variables can have types, as in $(p : Plane)$.

Table 3.1: Comparison of STRIPS and ADL languages.

- Disjunctive preconditions
- Conditional effects
- Universal quantification over dynamic universes (i.e., object creation and destruction)
- Domain axioms over stratified theories
- Specification of safety constraints

PDDL was originally developed by the AIPS-98 Competion Committee for use in defining problem domains. Since then, there have been several enhancements in the expression of the language.

The 2002 version (called PDDL2.1) [FL03] added many new features, mainly connected with adding time and objective functions to the language.

The 2004 version (called PDDL2.2) [EH03] added derived predicates and timed initial literals. The former are backward-chaining axioms that allow a planner to achieve a goal by making the antecedent of one of them true. The latter are literals that will become true at a predictable time independent of what the planning agent does.

The 2006 version (called PDDL3.0) [GL05] is the language for the 2006 competitions, which adds strong and soft constraints on plan trajectories

(i.e. constraints over possible actions in the plan and intermediate states reached by the plan), and strong and soft problem goals (i.e. goals that must be achieved in any valid plan, and goals desired to be achieved, but not necessarily), expressed in a restricted temporal logic.

Recently, a successor to PDDL, called Ontology with Polymorphic Types (OPT) has been proposed [McD05]. It is an attempt to create a general-purpose notation for expressing ontologies, definied as formalized conceptual frameworks for domain about which programs has to reason. Its syntax is based on PDDL, but it has a more elaborate type systems, which allows user to make use of higher-order constructs such as explicit λ-expressions.

3.2.3 Solving Planning Problems

Planning algorithms try to find valid plans for given planning problems. Planning can be viewed as graph search, either in the state space or in the plan space: *state-space planning* searches the directed graph whose nodes correspond to states and whose edges correspond to state transformations (i.e., actions), whereas *plan-space planning* searches the directed graph whose nodes correspond to (eventually incomplete) plans and whose edges correspond to plan transformations.

State-space Planning

The simplest way to build a planner is to cast the planning problem as search through the space of world states. Each node in the graph denotes a state of the world, and edges connect states that can be reached by executing a single action. When phrased in this manner, the solution to a planning problem (i.e., the plan) is a path through state-space. In other words, the objective of state-space planning is to find a path from the node that corresponds to the initial state to a node that corresponds to a goal state. The advantage of casting planning as a search problem is the immediate applicability of all the familiar brute force and heuristic search algorithms (see, for example [Kor87]).

Plan-space planning

Plan-space planning searches the directed graph whose nodes correspond to (eventually incomplete) plans and whose edges correspond to plan transformations. A plan is incomplete (and thus invalid) either if it is missing actions or if its actions are not completely ordered (*partial-order* planning). Partially ordered plans avoid unnecessary and potentially wrong ordering commitments between actions during planning (*least-commitment* planning). Plan transformations therefore often add actions or ordering constraints between actions to plans. The objective is to find a path from the nodes

that corresponds to the empty plan to a node that corresponds to a valid plan.

Progression and Regression Planning

The search of both state-space and plan-space planning can proceed *forward* from the initial state node to the goal nodes (*progression planning*) or *backward* from the goal nodes to the initial state node (*regression planning*). The formulation of planning problems as state-space search problems is as follows:

- The initial state of the search is the initial state from the planning problem. In general, each state will be a set of positive ground literals (literals not appearing begin false);
- The actions that are applicable to a state are all those whose preconditions are satisfied. The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals;
- The goal test checks whether the state satisfies the goal of the planning problem.

Forward search. From the earliest days of planning research (around 1961) until around 1998, it was assumed that forward state-space search was too inefficient to be pratical. The main reasons are that forward search does not address the irrelevant action problem (i.e., *all* applicable actions are considered from each state), and the approach quickly bogs down witout a good heuristic. In fact, the size of the search spaces often increases exponentially in the size of the planning problems. This implies that the graphs of typical STRIPS planning problems could not fit into the memories of computers and that finding even shortest action sequences could be computationally very hard. Therefore, planning techniques exploit the structure of planning problems in an attempt to find reasonably short action sequences for realistically sized planning problems.

One way to exploit structure is to focus the search with heuristics, often in the form of distance estimates to the goal vertices (HSP, UNPOP). Good distance estimates can be obtained automatically for state-space planning, for example from a data structure called a planning graph (FF, GRAPHPLAN, IPP, LPG, SGP, STAN, TGP).

A second way to exploit structure is to use knowledge about the structure of valid plans (TLPLAN).

A third way to exploit structure is to decompose planning problems into several subproblems that can be solved almost independently. Decomposing planning problems does not work well for puzzles like the Rubik's Cube but

seems to work well in domains in which humans plan well (everyday planning). For example, one can first identify the propositions that are part of the goal but not the start state, then find an action sequence for each of them that transforms the start state into a state that contains the proposition and finally merge the actions sequences. This is the main idea behind means-ends analysis, which picks each of the propositions in turn and first finds an action whose add list contains the proposition and then recursively tries to achieve all of those preconditions of the action that do not already hold in the start state. One can also decompose planning problems hierarchically by refining high(level actions to take them more concrete (hierarchical planning) (ABSTRIPS, DEVISER, FORBIN, NONLINE, O-PLAN, SHOP, SIPE).

Finally, a fourth way to exploit structure is to speed up planning for the current planning problem by utilizing information about how one has solved similar planning problems in the past (replanning or plan reuse). Case-based planning, for example, adapts plans from similar planning problems in the past to fit the current planning problem (CAPER, CAPLAN, CHEF).

Backward search. The main advantage of backward search is that it allows us to consider only relevant actions. An action is relevant to a conjunctive goal if it achieves one of the conjuncts of the goal. In addition that actions achieves some desidered literal, the actions must not undo any desidered literals. An action that statisfies this restriction is called consistent. Given a goal description G, let A be an action that is relevant and consistent. The general process of constructing predecessors for backward search is as follows:

- Any positive effects of A that apperar in G are deleted;
- Each precondition literal of A is added, unless it already appears.

Search termination occurs when a predecessor description is generated that is satisfied by the initial state of the planning problem. It is worth noting that neither forward search nor backward search is efficient without a good heuristic function.

3.3 Planning in complex domains

So far, we have described planning approaches for classical planning problems. These planning techniques find valid plans in the form of action sequences, ideally sequences of small lengths. Researchers have also studied far more complex planning pronlems, for example, where conditions have to be maintained rather than achieved, where the world changes even if the planning system does not execute actions, and where other systems are

present in either cooperative or competitive situations. This section hints at some approaches that can be used to cope with the mentioned issues.

Uncertainty about the outcomes of action executions gives rise to *decision-theoretic* planning problems. In case of nondeterministic actions, a number of successor states can result from their execution, and it cannot be predicted in advance which one will actually result. In case of states that are not totally partially observable, some relevant aspect of the world cannot always be observed (for example, due to sensor limitations) or cannot always be observed correctly (for example, due to sensor noise). In the presence of nondeterministic actions or partially observable states, planning systems cannot always know their current states but can estimate them, for example, in the form of sets of possible states or probability distributions over them. Then, planning systems often find plans that achieve the goal with high probability or, if the goal can be achieved for sure, minimize the number of action executions either in the worst case or on average. For the objective of minimizing the number of action executions in the worst case, planning techniques can draw on ideas from artificial intelligence search. For the objective of minimizing the number of action executions on average, planning techniques can draw on dynamic programming ideas from operations research (such as value iteration and policy iteration) to solve totally and partially observable Markov decision processes, which generalize graphs.

Totally observable Markov decision processes can model nondeterministic actions, while partially observable Markov decision processes can also model partially observable states. Decision-theoretic planning is computationally very hard, and decision-theoretic planning techniques thus exploit the structure of planning problems again, often by generalizing ideas from classical planning. For example, BURIDAN extends partial-order planning, SGP extends GRAPHPLAN, and MAXPLAN extends SATPLAN (see the last Section of this chapter).

Some planning systems rely on coercion to achieve the goal without sensing and thus continue to find action sequences (conformant or open-loop planning) [CGP]. However, it is not guaranteed that valid conformant plans exist for decision-theoretic planning problems or are of good quality because observations can now provide information about the current state. In general, one therefore often wants to find plans that contain sensing actions and select actions depending on the observations made during plan execution (conditional, contingent, or closed-loop planning) [WARPLAN-C]. For Markov decision processes it is sufficient to consider only conditional plans that map each state (for totally observable Markov decision processes) or each probability distribution over the states (for partially observable Markov decision processes) to the action that should be executed in it (policies), which reduces the search space and allows one to represent conditional plans compactly (reactive planning). The large number of possible contingencies makes planning for decision-theoretic planning problems

extremely time consuming.

One way of speeding up planning is to *interleave* planning and action executions [CONT-PLAN, SEP-PLAN], since action executions can result in additional observations, which can eliminate some contingencies and thus speed up planning. For example, instead of having to plan for all states that can result from the execution of an action, one can simply execute the action and then plan only for the successor state that actually resulted from its execution. There are different ways of interleaving planning and action executions. For example, agent-centered planning techniques find only the beginning of a valid plan by searching with a limited look-ahead around the current state, execute the plan, and repeat the process. Assumption-based planning techniques, on the other hand, find a plan that is valid provided their assumptions about the outcomes of action executions are correct. If these assumptions turn out not to hold during plan execution, they replan and repeat the process.

3.4 Planning by Abstraction

If a problem is sufficiently complex, then the dimension of state space could be enormous: even using new-generation planning algorithms or powerful heuristics could be not much help. Fortunately, many application contexts can be handled with an abstraction-based approach, by focusing the planner on the more difficult aspects of a problem, first. Once a sequence of more important subproblems has been detected, it is possible to separately solve each of them, then progressively add less important details, and finally grouping the results in the solution.

When applied to planning, this technique is called hierarchical planning. It employs one or more abstractions of a problem space to reduce the search. Instead of attempting to solve problems in the original problem space by plowing through the morass of details associated with a problem, a hierarchical planner first solves a problem in a simpler abstract space where the planner can focus on the real problem and ignore the details.

Systems that exploit an abstraction mechanism to solve planning problems are called hierarchical planners. Such mechanisms can be grouped into three categories, depending on the type of abstractions employed: abstract problem spaces, abstract operators, macro problem spaces. These three approaches are briefly described below, and compared in more detail in the next Chapter.

The first approach, hierarchical planning using abstract problem spaces, employs a hierarchy of abstract problem spaces to first solve a problem in an abstract space and then refine the abstract solution into successively more detailed spaces until it reaches the ground space. This type of hierarchical planning is sometimes called length-first hierarchical planning since a

problem is solved at one level of abstraction before moving to the next level.

The second approach, hierarchical planning using abstract operators, uses a predefined set of abstractions of the operators and expands each operator in the abstract plan to varying levels of detail. Instead of refining the entire plan at one level of detail, the planner refines the plan by selectively refining the individual operators in the plan. An operator is refined by replacing an abstract operator with a more detailed operator and achieving the unsatisfied preconditions of the new operator. This approach allows one part of the abstract plan to be expanded while another part is ignored, but eventually the entire plan will be expanded in the ground space. Unlike the length-first model, the abstractions need not to be a set of well-defined abstract problem spaces. Instead, the problem solver first selects abstract operators that directly achieve the goals and then refines the abstract operators by interting preconditions of the operators that must hold before operators can be applied in the ground space.

The third approach, abstract planning using macros, takes a problem and maps it into an abstract space defined by a set of macro operators and the solves the problem in the macro space. Unlike the first two approaches, once a problem is solved in the macro space, the problem is completely solved since the macros are defined by operators in the original problem space.

The work on this thesis builds on the third approach: in Chapter 8 a system to automatically generate macro-operators from static domain description is presented. The obtained macro-operators can be used, however, to fed the parametric hierarchical wrapper HW (described in Chapter 7, which is able to support all the three abstraction mechanisms. Actually, the presented approach can be considered as a hybrid between the second and the third approach. In fact, we do not consider abstract domains composed only by macro-operators. These are, instead, used as support for generating abstract operators.

3.5 Planning Systems: Literature Review

This section reviews the most important planning systems that have somehow marked the history of automated planning.

The first major planning system was STRIPS [FN71]. As described previously, it was designed as the planning component of the software for the Shakey robot project at SRI. Its overall control structure was based on that of GPS, the General Problem Solver [NS72], a state-space search system that used meansends analysis. STRIPS used a version of the QA3 theorem proving system as a subroutine for establishing the truth of preconditions for actions. However, the action representation used by STRIPS had more success than its algorithmic approach! In fact, almost all planning systems

have used a variant or an extension of the STRIPS language. Unfortunately, the proliferation of variants has made comparisons among planning algorithms very difficult. Moreover, there were limitations in representing complex problems. The Action Description Language (ADL) [Ped89] relaxed some of the restrictions in the STRIPS language and made it possible to encode more realistic problems. The Problem Domain Description Language (PDDL) [McD98] was introduced as a computer-parsable, standardized syntax for representing STRIPS, ADL, and other languages. PDDL has been used as the standard language for the planning competitions at the AIPS conference, beginning in 1998.

In the early 1970s planners generally worked with *totally ordered* action sequences. Problem decomposition was achieved by computing a subplan for each subgoal and then merging the subplans together in some order. This approach, called linear planning was soon discovered to be incomplete. It cannot solve some very simple problems, such as the Sussman anomaly found by Allen Brown during experimentation with the HACKER system. A complete planner must allow for interleaving of actions from different subplans within a single sequence. The notion of serializable subgoals [Kor87] corresponds exactly to the set of problems for which non interleaved planners are complete.

One solution to the interleaving problem was *goal regression* planning, a technique in which steps in a totally ordered plan are reordered so as to avoid conflict between subgoals. This was introduced by Waldinger [WAL77] and also used by Warren's WARPLAN [War76]. WARPLAN is also notable in that it was the first planner to be written in a logic programming language (Prolog) and is one of the best examples of the remarkable economy that can sometimes be gained by using logic programming: WARPLAN is only 100 lines of code, a small fraction of the size of comparable planners of the time. INTERPLAN [Tat74] also allowed arbitrary interleaving of plan steps to overcome the Sussman anomaly and related problems.

The ideas underlying partial-order planning include the detection of conflicts and the protection of achieved conditions from interference. The construction of partially ordered plans (then called task networks) was pioneered by the NOAH planner and by Tate's NONLIN system. Partial-order planning dominated the next 20 years of research, yet for much of that time, the field was not widely understood. TWEAK was a logical reconstruction and simplification of planning work of this time; his formulation was clear enough to allow proofs of completeness and intractability (NP-hardness and undecidability) of various formulations of the planning problem. Chapman's work led to what was arguably the first simple and readable description of a complete partial-order planner. An implementation of McAllester and Rosenblitt's algorithm called SNLP was widely distributed and allowed many researchers to understand and experiment with partial-order planning for the first time. Weld's group also developed UCPOP, the first planner

for problems expressed in ADL [PW92]. UCPOP incorporated the number-of-unsatisfied-goals heuristic. It ran somewhat faster than SNLP, but was seldom able to find plans with more than a dozen or so steps.

Although improved heuristics were developed for UCPOP [GS96], partial-order planning fell into disrepute in the 1990s as faster methods emerged. Nguyen and Kambhampati suggest that a rehabilitation is merited: with accurate heuristics derived from a planning graph, their REPOP planner scales up much better than GRAPHPLAN and is competitive with the fastest state-space planners. Avrim Blum and Merrick Furst revitalized the field of planning with their GRAPHPLAN system [BF95], which was orders of magnitude faster than the partial-order planners of the time. Other graph planning systems, such as IPP , STAN and SGP, soon followed. A data structure closely resembling the planning graph had been developed slightly earlier by Ghallab and Laruelle (1994), whose IXTET partial-order planner used it to derive accurate heuristics to guide searches. Nguyen et al. (2001) give a very thorough analysis of heuristics derived from planning graphs. The winner of the 2002 AIPS planning competition, LPG [GS02], searched planning graphs using a local search technique inspired by WALKSAT.

Planning as satisfiability and the SATPLAN algorithm were proposed by Kautz and Selman, who were inspired by the surprising success of greedy local search for satisfiability problems. Kautz et al. (1996) also investigated various forms of propositional representations for STRIPS axioms, finding that the most compact forms did not necessarily lead to the fastest solution times. A systematic analysis was carried out by Ernstetal. (1997), who also developed an automatic "compiler" for generating propositional representations from PDDL problems.

The BLACKBOX planner [KS98], which combines ideas from GRAPHPLAN and SATPLAN, was developed by Kautz and Selman. The resurgence of interest in state-space planning was pioneered by Drew McDermott's UNPOP program(1996), which was the first to suggest a distance heuristic based on a relaxed problem with delete lists ignored. The name UNPOP was a reaction to the overwhelming concentration on partial-order planning at the time; McDermott suspected that other approaches were not getting the attention they deserved.

Bonet and Geffner's Heuristic Search Planner (HSP) and its later derivatives [BG99] were the first to make state-space search practical for large planning problems. The most successful state-space searcher to date is Hoffmann's FASTFORWARD or FF [HN00], winner of the AIPS2000 planning competition. FF uses a simplified planning graph heuristic with a very fast search algorithm that combines forward and local search in a novel way.

Most recently, there has been interest in the representation of plans as binary decision diagrams, a compact description of finite automata widely studied in the hardware verification community. There are techniques for proving properties of binary decision diagrams, including the property of

being a solution to a planning problem. Cimatti et al. (1998) present a planner based on this approach. Other representations have also been used; for example, Vossen et al. (2001) survey the use of integer programming for planning. The jury is still out, but there are now some interesting comparisons of the various approaches to planning. Helmert (2001) analyzes several classes of planning problems, and shows that constraint-based approaches, such as GRAPHPLAN and SATPLAN are best for NP-hard domains, while search-based approaches do better in domains where feasible solutions can be found without backtracking. GRAPHPLAN and SATPLAN have trouble in domains with many objects, because that means they must create many actions. In some cases the problem can be delayed or avoided by generating the propositionalized actions dynamically, only as needed, rather than instantiating them all before the search begins.

Chapter 4

Abstraction

In artificial intelligence, *abstraction* is commonly used to account for the use of various levels of details in a given representation language or the ability to change from one level to another while preserving useful properties. Abstraction has been mainly studied in problem solving, theorem proving, knowledge representation (in particular for spatial and temporal reasoning) and machine learning. In such contexts, abstraction is defined as a mapping between formalisms that reduces the computational complexity of the task at stake.

This chapter focuses on abstraction techniques that can be exploited to improve the performances of automated classical planners, which allow implementing the problem-solving abilities of intelligent agents. First, section 4.2 illustrates the main categories under which traditional abstraction mechanisms are conceptually classified. Then, in section 4.3 a formal definition of abstraction hierarchy, together with its underlying elements, is given. Section 4.4 describes the main properties that characterize abstraction hierarchies. Finally, section 4.5 hints at the approaches useful to automatically generate abstraction hierarchies.

4.1 Introduction

Abstraction is known to be an effective speed-up technique for classical planners. The main idea is to reduce the search by focusing the problem on the more difficult aspects first. A hierarchy of abstract spaces is employed to first solve a problem in an abstract space, and then refine it into successively more detailed spaces until the ground space is reached. Actually, abstraction is often not effective on simple problems, due to the overhead introduced by the need of going back and forth across abstract spaces while performing the search. In other words, enforcing abstraction on simple problems may end up to wasting computational resources. On the other hand, the more planners will be used to solve problems of increasing complexity, like those

encountered in real-life applications, the more abstraction techniques will play a central role in the task of reducing the search time.

4.2 Abstraction Techniques

The most relevant abstraction techniques that have been proposed in the literature fall into three main categories:

- state-based
- action-based
- case-based

In the following, they will be described in detail.

4.2.1 State-based

State-based techniques exploit representations of the world at a lowel level of detail, and are mainly focused on removing predicates at different levels of granularity (either for preconditions only or for both preconditions and postconditions)[1]. These techniques do not consider abstraction on operators. Although they preserve the Upward Solution Property (USP) (see section 4.4 and [Ten88] for further details), their main drawback concerns the introduction of "false" solutions (i.e., not refinable solutions that anyway hold at the abstract level(s), due to the deletion of some constraints that apply to the ground level). Thus, the adoption of these techniques is strictly related to the actual ratio between "false" and "true" solutions [GW90], which must be kept reasonably low.

The most significant forms of state-based approaches rely on (a) relaxed models, obtained by weakening operators' applicability conditions [Sac74], and on (b) reduced models [Kno94], obtained by completely removing certain conditions from the problem space.

Relaxed Models

Relaxed models are constructed by removing preconditions of operators. This is the approach implemented in the ABSTRIPS planner [Sac74], where a criticality value is associated to each predicate, so that operators' preconditions can progressively be relaxed, while climbing the abstraction hierarchy, by dropping those predicates whose criticality value is under the one that characterizes the current level. It is worth noting that this approach does not alter operators' effects, which remain unchanged during the abstraction

[1]The overall technique can be classified as "a priori", abstractions being searched without resorting to information on solutions.

process. Viewed in terms of a state-space graph, the number of states in a relaxed model is the same as the initial model, but the possible transitions between the states is increased.

Reduced Models

In reduced models [Kno94] each predicate is associated with a unique level of abstraction according to the constraints imposed by the ordered monotonicity property; any such hierarchy can be obtained by progressively removing certain predicates from the domain (or problem) space. An abstract space is formed by dropping every instance of a particular set of predicates from both the states and the operators. Moreover, operators that only achieve predicates dropped from the abstract space are removed from the abstract space. In a reduced model, a single abstract state corresponds to one or more states in the original problem space. The language of a reduced model is a subset of the language of the original problem space.

This model has been adopted in ALPINE system (see [Kno91]), a system equipped with a hierarchical planner, able to automatically generate abstraction hierarchies.

It is worth noting that relaxed and reduced models are both homomorphisms of a problem space, which means that information is discarded in the process of constructing these models. As such, after a problem is solved in either type of abstract space, the abstract solution must be refined in the original space in order to ensure that the solution applies to the original problem.

4.2.2 Action-based

Action-based techniques use two different approaches: abstract operators based and macro-operator based.

The former uses a predefined set of abstractions of the operators and expands each operator in the abstract plan to varying levels of detail. Planners do not entirely refine plans at one level of detail, but they selectively refine the individual operators in the plan. An operator is refined by replacing an abstract operator with a more detailed operator and achieving the unsatisfied preconditions of the new operator. This approach allows one part of the abstract plan to be expanded while another part is ignored, but eventually the entire plan will be expanded in the ground space.

The latter takes a problem and maps it into an abstract space defined by a set of macro operators and then solves the problem in the macro space. In other words, a group of actions is combined to form macro-operators (e.g., [Ama68] and [Kor87]). Once a problem is solved in the macro space, the problem is completely solved since the macros are defined by operators in

the original problem space by a macro space.

Hierarchical Task Networks

The most significant example of operator-based approaches are Hierarchical Task Network (HTN) (e.g., [EHN94]), in which problem and operators are organized into a set of tasks: a high-level task can be reduced to a set of partially ordered, lower-level, tasks. Reductions allow specifying how to obtain a detailed plan from an abstract one.

4.2.3 Case-based

Case-based techniques are centered on a different perspective, assuming that a solution of a given problem can be found by adapting plans already found for similar problems. [2] Several different issues are very important in this framework: (i) how to define suitable metrics for measuring the similarity between problems, (ii) how to store and maintain a repository of "cases" encountered while solving problems, and (iii) which techniques and heuristics should be exploited to adapt a plan retrieved from the repository and deemed useful for solving the given problem. It is worth noting that the adoption of case-based planning is justified only agreeing with the conjecture that "repairing" an existing solution is computationally less costly than finding one from scratch, which is actually a very controversial issue.

In the literature, an interesting system that exploit these approach is the PARIS system [BW95], in which abstract planning cases are automatically learned from given concrete cases, although the user must provide explicit refinement rules between adjacent levels of the hierarchy.

Since, in this work, we are mainly concerned with action-based techniques, let us focus on the their pros and cons. The pionieristic work of Korf [Kor87] was not explicitly tailored for abstraction hierarchies -the adoption of macro-operators being limited to the ground level only. Although this choice shown to be useful on several domains (see also [BS04]), it negatively impacts on the average branching factor. On the other hand, this approach preserves both the soundness (macro-operators represent legal sequences of ground operators) and the completeness of the planner (none of the original ground operators being removed from the domain). As for the HTN-based techniques [EHN94], in a sense, they can be considered as a generalization of Korf's macro-operators, with a greater expressive power due to their capability of actually defining an abstraction hierarchy, together with the ability of allowing partial ordering among operators. The main drawback of this technique appears to be its strict dependence from the domain engineer,

[2] The overall technique can be classified as "a priori", abstractions being searched without resorting to information on solutions.

which is responsible for defining a (possibly) sound and complete HTN network for the given domain / problem. Furthermore, the amount of actual search strictly depends on the domain engineer's ability of devising high-level tasks with the desirable property of being easily put together to form a solution for the given problem.

4.3 Abstraction Hierarchies

In general, a planning domain can be defined in terms of two kinds of entities: *predicates* and *operators*. A particular kind of unary predicates can also be taken into account, giving rise to a third kind of entities -i.e., *types*- possibly organized according to a suitable "is-a" hierarchy.

To improve the performance of a planning algorithm, a domain can be organized into a set of abstraction levels, each of them containing its own set of predicates and operators. Thus, the original search space can be mapped into abstract spaces in which irrelevant details are disregarded at different levels of granularity. In particular, abstracting a domain leads to the definition of an abstraction hierarchy, consisting of a set of predicates and operators, together with a set of mapping functions devised to specify the mapping between two adjacent levels.

According to Giunchiglia and Walsh [GW90], an abstraction is a mapping between representations of a problem. In symbols, an abstraction

$$f : \Sigma_0 \rightarrow \Sigma_1$$

consists of a pair of formal systems (Σ_0, Σ_1) with languages Λ_0 and Λ_1 respectively, and an effective total function

$$f_0 : \Lambda_0 \rightarrow \Lambda_1.$$

Extending the definition, an abstraction hierarchy consists of a list of formal systems $(\Sigma_0, \Sigma_1, \ldots, \Sigma_{n-1})$ with languages $\Lambda_0, \Lambda_1, \ldots, \Lambda_{n-1}$ respectively, and a list of effective total functions

$$f_k : \Lambda_k \rightarrow \Lambda_{k+1},$$

$$(k = 0, 1, \ldots, n-2)$$

devised to perform the mapping between adjacent levels of the hierarchy.

Let us consider two abstraction levels, namely ground and abstract (the extension of the definitions to an n-level hierarchy being trivial).

A deterministic *ground operator* is characterized by a name, a list of parameters, and the specification of its pre- and post-conditions given in terms of ground predicates. A ground operator can be instantiated by substituting its parameters with objects taken from the given problem, thus giving rise to an instantiated ground operator (i.e., an *action*).

An *abstract operator* is characterized by a name, a list of parameters, and the specification of its overall pre- and post-conditions given in terms of abstract predicates.

A *macro-operator* is any legal sequence of ground operators, together with the specification of its overall pre- and post-conditions.

Formally, let ω be a sequence of operators (actions), a corresponding macro-operator (macro-action) can be defined by the following formulas:

$$\begin{cases} \gamma^+_\omega = \gamma^+_{\omega_1} \cup (\gamma^+_{\sigma'} \setminus \eta^+_{\omega_1}) \\ \gamma^-_\omega = \gamma^-_{\omega_1} \cup (\gamma^-_{\sigma'} \setminus \eta^-_{\omega_1}) \\ \eta^+_\omega = (\eta^+_{\omega_1} \setminus \eta^-_{\sigma'}) \cup (\eta^+_{\sigma'} \setminus \gamma^+_{\omega_1}) \\ \eta^-_\omega = (\eta^-_{\omega_1} \setminus \eta^+_{\sigma'}) \cup (\eta^-_{\sigma'} \setminus \gamma^-_{\omega_1}) \end{cases} \tag{4.1}$$

where γ^+, γ^-, η^+, and η^-, represent preconditions, negated preconditions, add-list, and delete-list of the resulting macro-operator, respectively.

Although -in principle- abstraction might be performed on both predicates (including types) and operators, there is no a predefined ordering in the abstraction process. In fact, one may start abstracting types, rather than predicates or operators -although any choice performed on one kind of mapping may impact on subsequent choices. Nevertheless, in this work we are mainly interested in abstracting operators starting from at least one supporting macro-operator, i.e., macro-operators whose pre- and post-conditions match the one defined for the corresponding abstract operator.

It is worth pointing out in advance that the easiest way to generate an abstract operator from a supporting macro-operator is to consider only the preconditions and the effects of the latter. Thus, in the following, the terms abstract- and macro-operator will be used as synonymous.

4.4 Formal Properties of Abstraction Hierarchies

To exploit and generate useful abstraction hierarchies, it is important to define their properties in terms of the relationships between the abstract levels and the ground level.

This section defines the properties of an abstraction hierarchy in terms of the relationship between an abstract plan and a concrete-level plan. This section first reviews two properties on abstraction hierarchy based on existence of planes: *"Does the existence of an abstraction solution guarantee the existence of a solution at any lower level? Or, is the converse true for an abstraction hierarchy?"*, then, it reviews properties which specificate a precise correspondence between solutions using a refinement relationship.

Tenemberg [Ten88] identified the upward and downward solution properties, which relate a ground space to an abstract space. The upward solution property is defined as follows:

Upward Solution Property: the existence of a ground-level solution implies the existence of an abstract-level solution.

Formally, given a n-level abstraction hierarchy, with level $n-1$ being the most abstract and level 0 the most concrete one. Let i be an integer between 0 and $n-2$.

Definition 1 (Upward Solution Property) *Whenever any i-th-level solution* Π_i *exists, there exists an abstract solution* Π_{i+1} *at level* $i+1$.

By this definition, if a ground-level solution Π_0 exists, then there exists a sequence of abstract solutions ending with Π_0, $\langle \Pi_{n-1}, \ldots, \Pi_0 \rangle$, such that each Π_i is and i-th level abstract solution. By the upward solution property, if there is no solution plan for a problem at an abstract level, then there is no solution at any lower level either. This fact follows directly from the upward solution property, since, if otherwise, a solution at any lower level would imply that solutions should exists at all higher levels, contradicting the initial assumption. This implication of the usp justifies a top-down refinement strategy when planning with an abstraction hierarchy. The inverse of the upward solution property is the downward solution property, which is defined as follows:

Downward Solution Property: the existence of an abstract-level solution implies the existence of a ground-level solution. In general, given a n-level abstraction hierarchy, with level $n-1$ being the most abstract and level 0 the most concrete one. Let i be an integer between 1 and $n-1$.

Definition 2 (Downward Solution Property) *Whenever any i-th-level abstract solution* Π_i *exists, there exists a solution* Π_{i-1} *at level* $i-1$.

By this definition, if an abstract solution Π_{n-1} exists, then there exists a sequence of abstract solutions ending with Π_0, $\langle \Pi_{n-1}, \ldots, \Pi_0 \rangle$, such that each Π_i is an i-th level solution. By the downward solution property, if any solution is found at an abstract level, then a ground-level solution exists for the original planning problem. Conversely, if there is no solution at the ground level, then no solutions exist at any higher level either. In other words, this property guarantees that no false solution at abstract level exist. Unfortunately, there are few abstraction spaces for which the downward solution property holds. Since an abstraction space is formed by dropping conditions from the original problem space, information is lost and operators in an abstract space can apply in situations in which they would not apply in the original space.

If the dsp does not hold, there is no guarantee that a refinement of the abstract solution exists. Since the downward solution property is too strong to guarantee, a set of weaker properties that constrain the refinement of an

abstract solution have been defined. The dsp and usp properties lack in the specification of a precise relationship between an abstract solution and a lower level solution. In fact, with usp, a ground-level solution Π_g implies the existence of an abstract solution Π_a, but it does not describe how Π_g is related to Π_a. A property that constrains the refinement of an abstract solution is called monotonic property, which is defined as follows:

Monotonic Property: the existence of a ground-level solution implies the existence of an abstract-level solution that can be refined into a ground-level solution while living the literals established in the abstract plan unchanged. The monotonic property states that if a solution exists it can be found without modifying an abstract plan in the process of refining that plan. In other words, an abstract solution should serve as an outline to a ground solution and thus should not be modified in the refinement process. In general, we can assume that there is a one-to-many relation *Refine* which, for any given abstract solution Π at level i, returns a set of solutions $\{\Pi_1, \Pi_2, \ldots, \Pi_B\}$ at level $i-1$. Let us assume that each plan in $Refine(\Pi)$ leaves all causal links of Π intact. During the transition from Π to Π_i, no plan steps and causal links are removed from Π. Then Π_i is called a *monotonic refinement* of Π. With the notion of monotonic refinement, we can now specify a variant of the upward solution property. Let i be an integer between 0 and $n-2$.

Definition 3 (Monotonic Property) *Whenever an i-th-level solution Π_i exists, there exists an abstract solution Π_{i+1} at level $i+1$, such that Π_i is a monotonic refinement of Π_{i+1}. In other words, $\Pi_i \in Refine(\Pi_{i+1})$.*

By this definition, if a ground-level solution Π_0 exists, then there exists a sequence of abstract solutions ending with Π_0, $\langle \Pi_{n-1}, \ldots, \Pi_0 \rangle$, such that each Π_i is an i-th level abstract solution, and every plan Π_{i-1} is a monotonic refinement of a previous plan Π_i.

The monotonicity property provides a criterion for backtracking that does not sacrifice completeness: if a problem solver will undo a literal established in an abstract plan while refining the plan, the system can backtrack to a more abstract level instead, since the property states that if a problem is solvable, an abstract solution exists that can be refined leaving the abstract plan unchanged. Tenenberg has shown that with precondition-elimination hierarchies upward solution property always holds, implying that the monotonic property holds, too. The monotonic property was implemented in the abstract planning system `ABTWEAK`. It was found that often excessive backtracking occurs due to violations of the abstract causal links. In this case, for many hierarchies, there is no improvement in planning efficiency over not using abstraction at all. While the monotonic property is useful for constraining the refinement process, it is rather weak. In fact, Knoblock [Kno91] proved that every abstraction space has this property, and introduced a restriction of it, called the ordered monotonicity property,

which is defined as follows:

Ordered Monotonicity Property: *Every* refinement of an abstract plan leaves *all* the literals that comprise the abstract space unchanged. To formally express the property, we need the definition of an ordered refinement:

Definition 4 (Ordered refinement) *A plan* Π_i *is an* ordered refinement *of an abstract plan* Π *if*

- Π_i *is a refinement of* Π*, and*
- *the new plan steps added into the abstract plan* Π *do not add or delete* any literals with a higher criticality value

Let i be an integer from 1 to $n-1$.

Definition 5 (Ordered Monotonic Property) *For every i-th-level solution* Π*, if* Π *has a refinement at level* $i-1$*, then* every *refinement of* Π *at level* $i-1$ *is an* ordered refinement *of* Π.

The ordered monotonicity property is more restrictive than the monotonicity property because it requires that not only there exists a refinement of an abstract plan that leaves the literals in the abstract plan unchanged, but every refinement of an abstract plan leaves all the literals in the abstract space unchanged. In general, not every hierarchy satisfies the ordered monotonic property. Moreover, despite the ordered monotonic property is very strong, backtracking across abstraction level may still occur. In other words, the ordered monotonic property is neither necessary nor sufficient for ensuring that backtracking never occurs across abstraction levels. In response to this issue, Baccus and Yang identified a property as the operationalized version of the downward solution property: the downward refinement property (DRP).

Definition 6 (Downward Refinement Property) *Every abstract solution at level i can be* monotonically refined *to a solution at the next lower level, level i-1.*

It is worth noting that if a hierarchy satisfies the DRP, then during planning, we only need to keep one copy of an abstract plan. From the DRP we know that this plan will lead to a correct plan at the next level. By induction, we also know that a ground-level solution will eventually be found. Thus, the DRP could be used to dramatically prune a search space without loss of completeness. Unfortunately, the DRP is not universal: it is not satisfied by every precondition-elimination hierarchy. Nor is it satisfied by every task-network hierarchy.

4.5 Automatic Generation of Abstractions

Although the use of abstraction hierarchy can significantly improve the performance of a planner, it is often difficult to find good abstractions, which often must be manually engineered by the designer of a domain. This process is largely a black art, since it is not even well-understood what makes a good abstraction.

The main work in this area has been done by Knoblock [Kno94], who provided algorithms for automatically generating abstraction hierarchies that are based on reduced models. In this thesis, a novel approach to generate automatically macro-operators, to be used for generating useful abstract-level descriptions, is described in chapter 8.

From a general perspective, macro-operators can be obtained by resorting to "a posteriori" or "a priori" analysis.

An "a posteriori" analysis can be done by processing solutions of previously-solved planning problems, under the assumption that solutions of planning problems often contain recurrent sequences of actions. The application of formula 4.1 to a sequence generates a macro-action that -by definition- leads to the same state that the given sequence of actions would achieve. In an "a posteriori" analysis, macro-actions must be uninstantiated to obtain macro-operators. To uninstantiate a macro-action, the objects involved in all its embedded actions are substituted by typed variables. A system able to perform an "a posteriori" analysis has been described in [AV01], where an adaptive mechanism that allows discovering relevant sequences from successful plans is proposed. Any such sequence becomes a candidate for generating abstract operators to be embedded into a hierarchical planner. To identify relevant sequences, a chunking technique that processes successful plans is exploited. Relevant sequences are identified by a feedforward neural network, fed by a vector of suitable metrics evaluated for each given sequence. A corresponding abstract operator is associated to each sequence, which is made available to the abstract level for any subsequent planning problem to be solved. Due to the dependency between abstract operators and already-solved planning problems, an agent equipped with such algorithm may exhibit an individual adaptation to the given environment.

An "a priori" analysis is performed by processing the given planning domain (and the problem, if needed), without resorting to plans previously found (see for example, [ACV03a]). Chapter 8 describes the DHG system, one of the contribution of this thesis, devised to automatically generate macro-operators starting from a from a ground-level description of a domain, to be used for generating useful abstract-level descriptions.

Analyzing the relationships among the operators of the domain, a set of relevant sequences can be identified and used for building suitable macro-operators. Given a sequence of operators, a corresponding macro-operator can be defined that embeds the sequence, and whose preconditions and ef-

fects can be evaluated according to formula 4.1. Since the parameters of a macro-operator are in fact variables, generating pre- and post-conditions of the resulting operator involves a variable-unification process, which may led to semantic inconsistencies. For instance, this problem may occur when dealing with "position predicates", `(at ?o - object ?l - location)` taken from the *Logistics* domain. According to its intended semantics, there cannot be two predicates stating that the same object is in two different locations. This condition, not explicitly stated in the domain description, can be expressed through the use of suitable *state invariants*. A detailed description about how to find state invariants is given in [FL98], where four kinds of state invariants are defined: identity, state membership, uniqueness of state membership, and fixed resource. The information about the domain, enriched with invariants, allows to discriminate between different alternatives, so that macro-operators' parameters can be correctly unified.

Part II

A Personalized Adaptive Cooperative MultiAgent System

Chapter 5

The PACMAS Architecture

The information available on the WWW is continuously growing from different points of view: information sources are increasing, topics discussed are becoming more and more heterogeneous, and stored data has reached a considerable size. It has become a difficult task for Internet users to select contents according to their personal interests, especially if contents are continuously updated (e.g., news, newspaper articles, reuters, rss feeds, blogs, etc.). Unfortunately, traditional filtering techniques based on keyword search are often inadequate to express what the user is really searching for. Furthermore, users often need to refine by hand the achieved results.

Supporting users in handling with the enormous and widespread amount of web information is becoming a primary issue. To this end, an automated system able to retrieve information from the Internet, and to select the contents really deemed relevant for the user, will be described along this chapter.

This chapter presents PACMAS (which stands for "Personalized Adaptive Cooperative MultiAgent System"), a novel multiagent architectures aimed at retrieving, filtering and reorganizing information according to users' personal interests [ACMV05]. After a brief introduction, section 5.2 describes the architecture from a "social level" point of view, by illustrating the characteristics of agents acting as a group. Then, section 5.3 focuses on the characteristics of agent as stand-alone entities.

5.1 Introduction

In the literature, software agents have been widely proposed for retrieving information from the web (see for example [KAH94] [EW95b] [Kra97]). Furthermore, several machine learning techniques have been applied to text categorization (see [Yan99] for a detailed comparison).

Assuming that information sources are a primary operational context for software agents, the following categories can be identified focusing on their

specific role:

- *information agents*, able to access to information sources and to collect and manipulate such information [Mae94];
- *filter agents*, able to transform information according to user preferences [LKRH90];
- *task agents*, able to help users to perform tasks by solving problems and exchanging information with other agents [GSF+04];
- *interface agents*, in charge of interacting with the user such that she/he interacts with other agents throughout them [Lie97];
- *middle agents*, devised to establish communication among requesters and providers [DSW97].

Although this taxonomy is focused on a quite general perspective, alternative taxonomies could be defined focusing on different features. In particular, one may focus on capabilities rather than roles, a software agent being able to embed any subset of the following capabilities:

- *autonomy*, to operate without the intervention of users;
- *reactivity*, to react to a stimulus of the underlying environment according to a stimulus/response behaviour;
- *pro-activeness*, to exhibit goal-directed behavior in order to satisfy a design objective;
- *social ability*, to interact with other agents according to the syntax and semantics of some selected communication language;
- *flexibility*, to exhibit reactivity, pro-activeness, and social ability simultaneously [WJ95];
- *personalization*, to personalize the behavior to fulfill user's interests and preferences;
- *adaptation*, to adapt to the underlying environment by learning how to react and/or interact with it;
- *cooperation*, to interact with other agents in order to achieve a common goal;
- *deliberative capability*, to reason about the world model and to engage planning and negotiation, possibly in coordination with other agents;
- *mobility*, to migrate from node to node in a local- or wide-area network.

Table 5.1: Capabilities of software agents.

Capability	**Focus on the ability of ...**
Autonomy	Operating without the intervention of users.
Reactivity	Reacting to a stimulus of the underlying environment according to a stimulus/response behaviour.
Proactivity	Exhibiting goal-directed behavior in order to satisfy a design objective.
Social ability	Interacting with other agents according to the syntax and semantics of some selected communication language.
Flexibility	Exhibiting reactivity, proactiveness, and social ability simultaneously [WJ95].
Personalization	Personalizing the behavior to fulfill user's interests and preferences.
Adaptation	Adapting to the underlying environment by learning how to react and/or interact with it.
Cooperation	Interacting with other agents in order to achieve a common goal.
Deliberative ability	Reasoning about the world model and of engaging planning and negotiation, possibly in coordination with other agents.
Mobility	Migrating from node to node in a local- or wide-area network.

Table 5.1 briefly depicts such capabilities and the corresponding focus.

The following sections describe a generic multi-agent architecture designed to support the implementation of applications aimed at: (i) retrieving heterogeneous data spread among different sources (i.e., generic html pages, news, blogs, forums, and databases), (ii) filtering and organizing them according to personal interests explicitly stated by each user, and (iii) providing adaptation techniques to improve and refine throughout time the profile of each selected user.

Each agent is autonomous and flexible, and may implement (one or more of) the following capabilities: personalization, adaptation, and cooperation. The overall architecture has been called PACMAS, being designed to support the implementation of Personalized, Adaptive, and Cooperative MultiAgent Systems. The PACMAS architecture can easily give rise to specific systems by (1) identifying the characteristics of the data flow that occurs from information sources to users (and vice versa), and (2) customizing each involved agent according to its actual role and capabilities.

5.2 Macro-architecture

PACMAS is a generic multiagent architecture aimed at retrieving, filtering and reorganizing information according to users' interests. PACMAS agents can be personalized, adaptive, and cooperative, depending on their specific role.

The overall architecture (depicted in Figure 5.1) encompasses four main levels (i.e., information, filter, task, and interface), each being associated to a specific role. The communication between adjacent levels is achieved through suitable middle agents, which form a corresponding mid-span level.

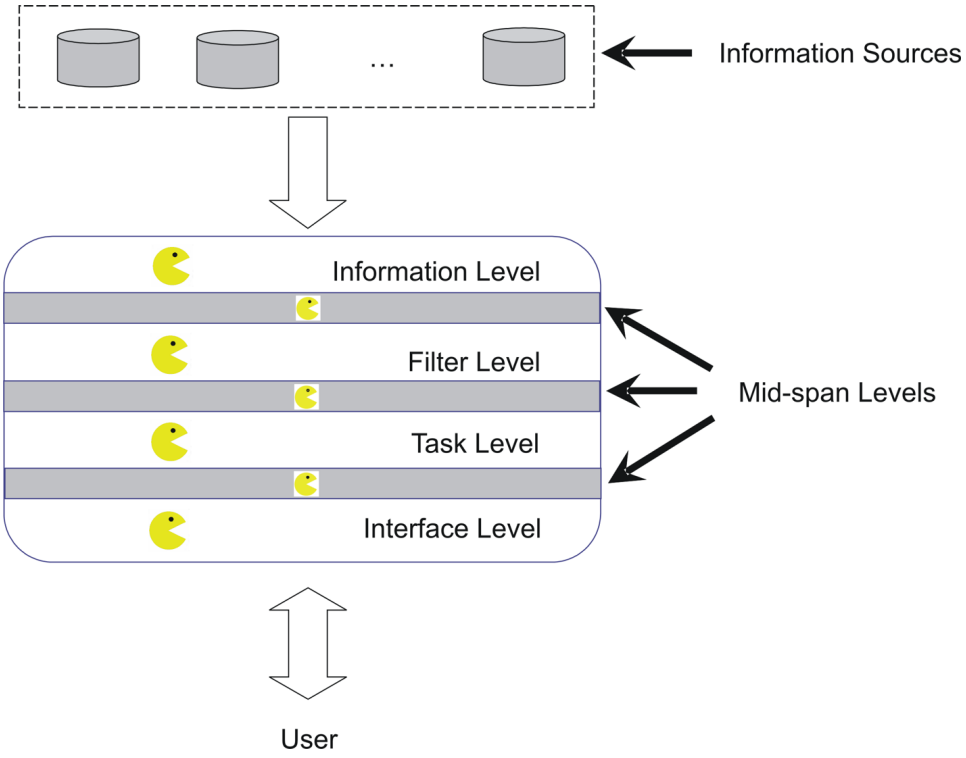


Figure 5.1: The PACMAS Architecture.

Each level is populated by a society of agents, so that communication may occur both horizontally and vertically. The former kind of communication supports cooperation among agents belonging to a specific level, whereas the latter supports the flow of information and/or control between adjacent levels through suitable middle-agents.

5.2.1 Information Level

At the information level, agents are entrusted with extracting data from the information sources. Each information agent is associated to one information

source, playing the role of wrapper. Upon extraction, the information is then made available to the underlying filter level.

5.2.2 Filter Level

At the filter level, agents are aimed at selecting information deemed relevant to the users, and cooperate to prevent information from being overloaded and redundant. Two filtering strategies can be adopted: generic and personal. The former applies the same rules to all users; whereas the latter is customised for a specific user. Each strategy can be implemented through a pipeline of filters, since data undergo an incremental refinement process. The information filtered so far is then made available to the task level.

5.2.3 Task Level

At the task level, agents arrange data according to users' personal needs and preferences. In a sense, they can be considered as the core of the architecture. In fact, they are devoted to achieve users' goals by cooperating together and adapting themselves to the changes of the underlying environment. In general, they can be combined together according to different connection modes, depending on the specific application.

5.2.4 Interface Level

At the interface level, a suitable interface agent is associated to each different user interface. In fact, a user can generally interact with an application through several interfaces and devices (e.g., pc, pda, mobile phones, etc.). Interface agents usually act individually without cooperation. On the other hand, they can be personalized to display only the information deemed relevant to a specific user. Moreover, in complex applications, they can adapt themselves to progressively improve their ability in supplying information to the user.

5.2.5 Mid-span Level

At the mid-span level, agents are aimed at establishing communication among requesters and providers. In the literature, several solutions have been proposed: e.g., blackboard agents, matchmaker or yellow page agents, and broker agents (see [DSW97] for further details). In the PACMAS architecture, agents at the mid-span level can be implemented as matchmakers or brokers, depending on the specific application.

5.3 Micro-architecture

Keeping in mind that agents may be classified along several ideal and primary capabilities that they should embed, let us first recall the agent taxonomy proposed in [Nwa96]. In such taxonomy, three primary capabilities have been identified: *autonomy*, *learning*, and *cooperation* (see Figure 5.2-a). In our view, agents are always autonomous and flexible, hence we deem that autonomy should not be explicitly listed in a diagram. On the contrary, we claim that personalization should be taken into account as a primary feature while depicting the characteristics of software agents. The resulting taxonomy, which considers *personalization*, *adaptation*, and *cooperation* as primary capabilities, is depicted in Figure 5.2-b.

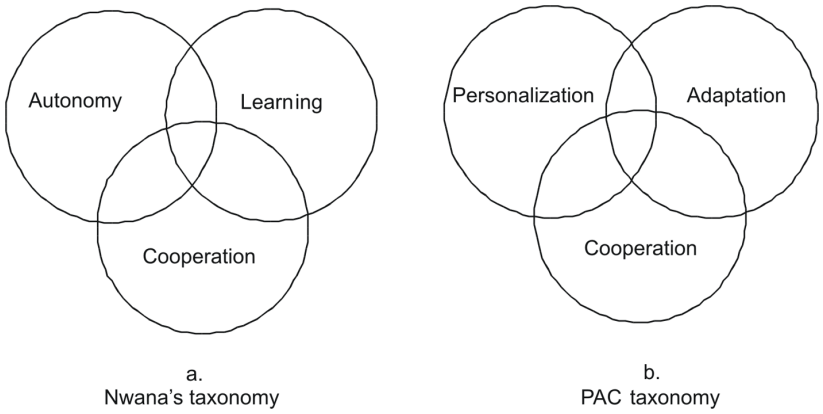


Figure 5.2: Agents Taxonomies.

5.3.1 Personalization

As for personalization, an initial user profile is provided in form of a list of keywords, representing users' interests. The information about the user profile is stored by agents belonging to the interface level. It is worth noting that, to exhibit personalization, filter and task agents may need information about the user profile. This flows up from the interface level to the other levels through the middle-span levels. In particular, agents belonging to mid-span levels (i.e., middle agents) take care of handling synchronization and avoiding potential inconsistencies. Moreover, the user behavior is tracked during the execution of the application to support explicit feedback, in order to improve her/his profile.

5.3.2 Adaptation

As for adaptation, different techniques may be employed depending on the application to be developed. The model adopted in the case studies de-

scribed in chapter 6 is based on the concept of "mixtures of experts". Each expert is implemented by an agent able to select relevant information according to an embedded string of feature-value pairs, features being selectable from an overall set of relevant features defined for the given application. The decision of adopting a subset of the available features has been taken for efficiency reasons, being conceptually equivalent to the one usually adopted in a typical GA-based environment [Gol89], which handles also dont-care symbols. Beginning with an initial population of experts, the system evolves by the creation of further experts according to covering, crossover, or mutation mechanisms.

5.3.3 Cooperation

As for cooperation, agents at the same level exchange messages and/or data to achieve common goals, according to the requests made by the user. Cooperation is implemented in accordance with the following modes: centralized composition, pipeline, and distributed composition (see Figure 5.3). In particular: (i) centralized compositions can be used for integrating different capabilities, so that the resulting behavior actually depends on the combination activity; (ii) pipelines can be used to distribute information at different levels of abstraction, so that data can be increasingly refined and adapted to the user's needs; and (iii) distributed compositions can be used to model a cooperation among the involved components aimed at processing interlaced information. The most important form of cooperation concerns

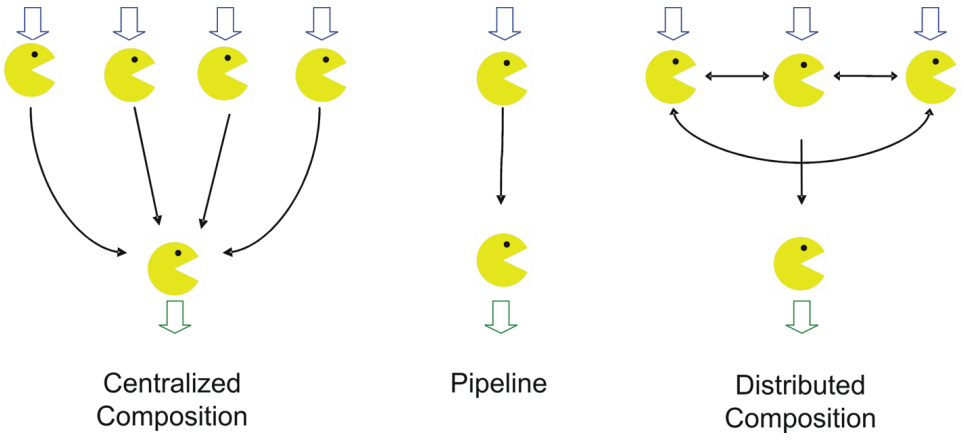


Figure 5.3: Agents Connections.

the "horizontal" control flow that occurs between peer agents. For instance, filter agents can interact in order to reduce the information overload and redundancy, whereas task agents can work together to solve problems that require social interactions to be solved.

Chapter 6

Case Studies

This chapter illustrates two applications that have been implemented exploiting the PACMAS architecture (which is described in the previous chapter). The first one is focused on giving a support to undergraduate and graduate students in their university activities; the second one is devoted to create press-reviews from online newspapers through the classification of newspaper articles. Despite the apparent differences, the two applications have in common several issues: information retrieval and extraction, information filtering, information processing, and results presentation. Therefore, they can be considered as interesting case studies able to highlight the peculiarities of the PACMAS architecture. Both the proposed case studies have been implemented using Jade [BPR00] as the underlying framework to support agents' mobility and communication functionalities.

6.1 Case Study 1: Supporting Students in University Activities

This case study is focused on giving a support to undergraduate and graduate students in their university activities [1].

6.1.1 Motivation

Let us consider a typical University Department. It generally makes available the information about courses, seminars, exams, professors, and students on different areas, e.g.: web sites, forums, and news (NNTP) servers. All the relevant information is not directly available but it is usually spread on the department portal, on the web site of each course, and on the personal page of each professor. Moreover, each professor might activate her/his own

[1] This work has been partially funded by the Italian Ministry of University and Research under the program *Programmi di Ricerca Scientifica di Rilevante Interesse Nazionale* (PRIN 2003).

news and forum services. Some of the information potentially interests all students, such as lesson timetables, exams' dates, taxes, and student tutoring. On the other hand, students belonging to different courses are obviously interested in different lessons and exams. For example, a student attending the MSc in Computer Science may be interested in the *Object Oriented Programming Languages I* course rather than in the *Processors and Embedded Systems Architectures* one. Similarly, a student attending the MSc in Digital Microelectronics may be interested in the *Processors and Embedded Systems Architectures* course rather than in *Object Oriented Programming Languages I* one. Typically, a student in search of relevant information about her/his University activities browses web sites and reads announcements from forum and news services. This is often a repetitive and boring task that can be automated. From our perspective, personalization and adaptation represent the added value of such an automated system.

6.1.2 Implementation

In order to provide an e-service able to support undergraduate and graduate students in their university activity at the Department of Electrical and Electronic Engineering (DIEE) of the University of Cagliari, a prototype of an e-service has been developed. This work is part of a wider network of multiagent systems, described in [GAV05]. It is built upon the PACMAS architecture, and exploits the JADE framework for supporting agent mobility and communication functionalities. Let us note that supporting students involves several activities: information retrieval and extraction, information filtering, information processing, and results presentation. Each activity is attained by exploiting a suitable level of the PACMAS architecture.

Information Retrieval and Extraction

It is carried out at the information level by a set of information agents, devoted to process information sources. Each agent plays the role of wrapper, and it is specialized for dealing with a specific information source: e.g., web pages, forums or news services. In the current implementation, information agents are not personalized, not adaptive, and not cooperative ($\overline{PAC}$). Personalization is not supported, since information agents are aimed at retrieving information potentially relevant to all students, regardless of their personal interests and preferences. Adaptation is also not supported, being the system mainly concerned with changes in users needs rather than in the underlying environment[2]. Cooperation is also not supported, cause each information agent is devoted to wrap a different information source.

[2]In this particular case the variability of the information sources

Information Filtering

It is carried out at the filter level by a set of filter agents. In particular, this level contains a set of redundancy filter agents (one for each information source), an anti-spam filter agent and a population of personal filter agents (one for each user of the system). Redundancy filters cooperate together to remove the redundancy of data provided by the information sources (throughout the information agents). Redundancy filters are not personalized, not adaptive, and cooperative ($\overline{PA}C$). Similarly to information agents, personalization and adaptation are not required. On the other hand, cooperation is required to prevent the information from being redundant. The anti-spam filter is not personalized, not adaptive, and not cooperative ($\overline{PAC}$)[3]. Being not dependent from a specific student, it filters the same information by removing undesirable contents according to a rule-based mechanism. Personal filters are personalized, adaptive and not cooperative ($PA\overline{C}$). As for personalization, they are sensible to any explicit change imposed by the corresponding student or to a change that occurs in the curriculum of the student. As for adaptation, they are able to progressively adapt their filtering capabilities according to the choices performed by the corresponding student during the lifetime of the agent. Cooperation is not supported; in fact, in the current release of the system, only a specific support for implementing voting policies according to the guidelines of GA-based systems is supplied.

Information Processing

It is carried out at the task level, where agents are devoted to perform different tasks according to the requirements imposed by the corresponding user. In particular, each task agent is customized for a specific task (e.g., lessons timetable, seminars, and exams scheduling). To process information regarding exams scheduling, the agent *ExamAgent* is actived, while the *CourseAgent* is devoted to manage information about courses in general.

Agents belonging to the task level exploit a model centered on the concept of "mixtures of experts", each expert being implemented by an agent. The system supports each user with a specific population of experts, handled in accordance with the basic guidelines of online systems, expecially the ones that characterize evolutionary environments. Task agents are personalized, adaptive, and cooperative (PAC). Personalization is required since different behaviors are associated to different students. Adaptation is required since they adapt themselves to the needs of the corresponding student through a GA-based feedback mechanism. Cooperation is required since they usually need other task agents to successfully achieve their own goals.

[3]In the current release of the system anti-spam agents are not permitted to implement adaptation, although in principle this property may be supplied in a future release.

Figure 6.1: JSP graphical interface for student support system.

Results Presentation

It is carried out at the interface level, through agents aimed at interacting with the users. Agents and users interact through a suitable graphical interface that can be run on several devices, including mobile phones. A different interface agent has been associated to each device. In the current implementation, the system embodies a graphical interface that runs on several devices, including MIDP 1.0 compliant devices (as the one shown in Figure 6.2), and JSP web pages (see Figure 6.1)[4].

Interface agents are also devoted to handle user profile and propagate it by the intervention of middle agents. Furthermore, any feedback provided by the user can be exploited by the adaptive mechanism to improve the user profile.

Interface agents are personal, adaptive, and not cooperative ($PA\overline{C}$). Personalization is required to allow each student the customization of her/his interface. Adaptation is supported, since an interface agent must adapt to the changes that occur in the preferences and interests of the correspond-

[4]Available at: http://iascw.diee.unica.it/PacmasWWW

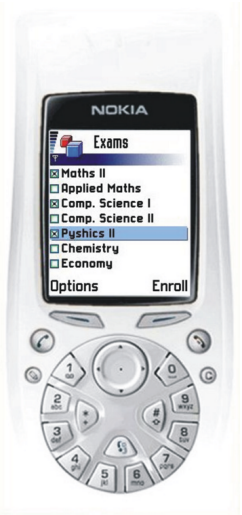


Figure 6.2: User interface on a MIDP compliant device.

ing student. Cooperation is not supported by agents that belong to this architectural level.

Agents at middle level are implemented as brokers. There are three broker agents that manages communication between adjacent levels: *BrokerTaskInterface*, *BrokerFilterTask*, and *BrokerInformationFilter*.

Table 6.1 summarizes the capabilities of the adopted agents, together with the corresponding activity, for each level of the PACMAS architecture.

Table 6.1: Agents for supporting students.

Level	**Activity**	**Agents**	**Capabilities**
Information	information retrieval and extraction	*DBWrapper*	$\overline{PAC}$
		NewsWrapper	$\overline{PAC}$
		ForumWrapper	$\overline{PAC}$
		SiteWrapper	$\overline{PAC}$
Filter	information filtering	*Redundancy*	$\overline{PAC}$
		Anti-spam	$\overline{PAC}$
		Personal	$P\overline{AC}$
Task	information processing	*ExamAgent*	PAC
		CourseAgent	PAC
Interface	input handling and results presentation	*WebInterface*	$P\overline{AC}$
		PhoneInterface	$P\overline{AC}$
Middle	handling interations	Brokers	$\overline{PAC}$

6.1.3 Experiments and Results

A prototype of the case study has been implemented as a web service. It has been tested on the information system of the Department of Electrical and Electronic Engineering (DIEE) at the University of Cagliari. The system is able to retrieve information from a set of specific webpages and forums, and filters and shows only the information deemed relevant by the associated user. The system is also able to learn and refine the user profile to better satisfy its requests. Preliminary tests on the system have shown that the approach is effective, and the system was able to succesfully tackle with a number of user connected contemporarily. A beta version of the web service is available at: *http://iascw.diee.unica.it/PacmasWWW*.

6.2 Case Study 2: Newspaper Articles Classification

This section describes how the generic architecture has been customized to implement a prototype of the system devised to perform text categorization [CDMV05][5]. In the following, we illustrate how each level of PACMAS supports the implementation of the proposed application.

6.2.1 Motivation

All the information sources belonging to the WWW make it hard for users to choose the most suitable according to their interests. Finding useful information of personal interest has become difficult for Internet users. Ideally, users should be able to take advantage of the wide range of available information while being able to find the one she/he is interested in. In particular, manually selecting newspaper articles is quite difficult or not feasible within the time constraints common for most users also considering that the results could not perfectly fit with the user interests. Some systems try to perform that task automatically, performing content-based filtering. In particular, software agents have been widely proposed for retrieving information from the web ([SM03], [Lie95], and [CCGJ04]).

6.2.2 Related Work

In the following, some related work on agent-based information retrieval is briefly recalled and the text categorization problem is illustrated.

[5]This prototype is one of the possible implementations that can be developed starting from the proposed generic architecture.

Agent-based Systems for Information Retrieving

Several multiagent systems have been proposed to support the user in the task of retrieving information from the web. Among them let us recall NewT [SM03], Letizia [Lie95], WebWatcher [AFJM95], and SoftBot [EW95b].

NewT [SM03] is designed as a collection of information filtering interface agents. Interface agents are intelligent and autonomous computer programs, which learn users' preferences and act on their behalf. This system uses a keyword-based filtering algorithm. The learning mechanisms used are relevance feedback and genetic algorithms.
Letizia [Lie95] is a user interface agent that assists a user browsing the World Wide Web. The model adopted by this system is that the search for information is a cooperative venture between the human user and an intelligent software agent. Letizia and the user both browse the same search space of linked web documents, looking for "interesting" ones.

WebWatcher [AFJM95] is an information search agent that follows web hyperlinks according to users' interests, returning a list of interesting links to the user.

In contrast to systems for assisted browsing or information retrieval, the SoftBot [EW95b] accepts high level user goals and dynamically synthesizes the appropriate sequence of Internet commands using a suitable ad-hoc language to satisfy those goals.

Finally, let us point out that current web search engines basically rely only on purely syntactical textual information retrieval. There are only a few approaches that try to integrate a set of different and specialized sources, but unfortunately it is very difficult to maintain and to develop this kind of systems [Kno94].

Text Categorization

The main goal of text categorization is to classify documents into a set of predefined categories. Each document can be in multiple or exactly one category. Using machine learning, the objective is to learn classifiers from examples, which perform the category assignments automatically, according to a supervised learning approach.

A major characteristic, or difficulty, of text categorization problems is the high dimensionality of the feature space. The native feature space consists of the unique terms (words or phrases) that occur in documents, which can be tens or hundreds of thousands of terms, even for a moderate-sized text collection. This is prohibitively complex for many learning algorithms. Thus, the first step in text categorization is to transform documents into a representation suitable for the underlying learning algorithm and the classification task.

Typycally, after counting the number of occurrences of a word `w` in a

document –giving rise to an unordered *bag of words* [ADW94]– suitable stemming algorithms [Por80] are applied to avoid unnecessarily large feature vectors. Each distinct word stem w_i corresponds to a feature, with the number of occurrences (in the entire document) of the word w_i as value. Words are considered as features only if they occur in the training data at least a predefined number of times except when they are considered as *stop-words* (like *and*, *or*, *is*, etc.).

To further reduce the number of considered terms, suitable feature selection methods can be applied. Automatic feature selection methods include the removal of non-informative terms according to corpus strategies, and the construction of new features which combine lower-level features (i.e., terms) into higher-level orthogonal dimension. Among different feature selection methods, let us recall document frequency, information gain, mutual information, a χ^2 statistic, and term strength (see [YP97] for a detailed comparison among them).

After selecting the terms, for each document a feature vector is generated, whose elements are the feature values of each term. A commonly used feature value is the `TF` (Term Frequency) `x IDF` (Inverse Document Frequency) measure.

Among machine learning techniques applied to text categorization, let us cite multivariant regression models [YC94], kNearest Neighbor classification [YL99], Bayes probabilistic approaches [TH93], decision trees [LR94], neural networks [EW95a], symbolic rule learning [MRG96] and inductive learning algorithms [CS96].

6.2.3 Implementation

In the following, we illustrate how each level of the architecture supports the implementation of the proposed application.

Information Level

At the information level, agents play the role of wrappers, each one being associated to a different information source. The agents at this architectural level are devoted to perform the information extraction. In the current implementation, two formats of Internet sources are supported, i.e. *RSS* and *HTML/XHTML*, each of them wrapped by a different agent. The *RSS-Wrapper* agent extracts information from online newspapers in RSS format, containing news articles. The *HTMLWrapper* agents extracts information by directly parsing Internet Pages in HTML format. RSS (Really Simple Syndication) is a well-structured format and it is very simple to be processed. On the contrary, HTML is often bad-formed and so needs ad-hoc algorithms to be correctly parsed. Another agent, the *IPTCWrapper*, is devoted to wrap the adopted "generic" taxonomy that is a subset of the one proposed by the

Figure 6.3: A fragment of the adopted (italian) taxonomy and its english translation.

International Press Telecommunications Council [6] (a fragment is depicted in Figure 6.3).

Information agents are not personalized, can be adaptive or not, and not cooperative (shortly $\overline{PAC}$ or $\overline{PAC}$). Personalization is not supported at this level, since information agents are only devoted to wrap information sources, which are user-independent. Adaptation is not supported by the *RSSWrapper*, since we assume that the structure of the information sources (being the *RSS* format a definitive standard) do not vary, and are user-independent. On the other hand, the *HTMLWrapper*, supports a simple user-assisted adaptative behaviour, since web sites often change in structure, making difficult the automatic adaptation of the systems. Cooperation is also not supported by the information agents, since they retrieve information from different sources, each of them having a specific role in the chosen application.

Filter Level

At the filter level, a population of agents manipulates the information belonging to the information level through suitable filtering strategies. First, a set of filter agents removes all the non-informative words such as prepositions, conjunctions, pronouns and very common verbs by using a standard stop-word list. After removing the stop words, a set of filter agents performs a stemming algorithm [Por80] to remove the most common morphological and inflexional suffixes from all the words. Then, for each class, a set of filter agents selects the features relevant to the classification task, according

[6] http://www.iptc.org/

to the information gain method. Let us recall that information gain measures the number of bits of information obtained for category prediction by knowing the presence or absence of a term in a document.

Filter agents can be personalized or not, whilst they are not adaptive and they are cooperative (shortly $\overline{PA}C$ or $P\overline{A}C$). Personalization is supported by the *StopWordsRemover* and *Stemmer* agents, since the corresponding filtering algorithms depend on the language of the texts processed. Thus, if a user of the systems needs to process articles in a specific language, the filter agents can be personalized accordingly [7]. The *FeatureSelector* agent is instead not personalized, since its filtering strategy is language-independent (and therefore user-independent). Adaptation is not supported at the filter level, since all the adopted filtering strategies do not change during agents activities. Cooperation is supported by all the filter agents, since they cooperate continuously in order to perform the filtering activity.

Task Level

At the task level, a population of agents is devoted to perform the classification activities. Each task agent embodies a classifier specialized for handling one specific class with one specific algorithm. In the current release, it is possible to choose between two classification algorithms: k-NN and weighted k-NN, altough -in principle- the system has been designed to support any algorithm. The k-nearest neighbour is a classification method based upon observable features. The algorithm selects a set which contains the k nearest neighbours and assigns the class label to the new data point based upon the most numerous class with the set. The weighted k-nearest neighbour is a variant version of k-NN that weights the voting strength of a case to be classified into a category, through a distance function (see [CS93]). All the involved agents are trained in order to recognize a specific class. Given a document in the test set, each previously trained agent, through its embedded classifier, ranks its nearest neighbours among the training documents to a distance measure[8], and uses the most frequent category of the k top-ranking neighbours to predict the categories of the input document. Each task agent is also devoted to measure the classification accuracy according to the confusion matrix [KP98].

Furthermore, agents at this architectural level are also devoted to performing classification according to users preferences automatically composing topics. Composition has been performed through the cooperation of the involved task agents. For instance, the "compound topic" politics *and* economy is obtained by the cooperation of the task agent expert in recognizing *politics* together with the task agent expert in recognizing *economy*.

[7] In the current release of the systems only the italian and english languages are supported.

[8] In the current implementation the cosine-distance measure has been adopted.

Moreover, a number of task agents (the *FeedBack* agents) are devoted to deal with the feedback provided by the user that flows up from the interface level. In the current implementation, such agents embody the K-NN initially trained with a set of examples classified as *of-interest*. The training process is repeated/updated when the amount of feedback trespasses a given threshold.

Task agents are personalized, adaptive, and cooperative (shortly PAC). Personalization is supported at this level, since agents at this architectural level perform the classification taking into account users needs and preferences. Adaptation is supported by the task agents since through the feedback mechanism they continuously adapt themselves to the user. Cooperation is supported by the task agents, since agents have to interact each other in order to achieve the classification task (expecially during classification on multiple categories).

Interface Level

At the interface level, agents are aimed at interacting with the user. In the current implementation, agents and users interact through a suitable graphical interface that runs on a pc (see Figure 6.4). Interface agents are also devoted to handle user profile and propagate it by the intervention of middle agents. By the interacting with the interface agent, the user can set her/his preferences. In particular, s/he can set preferences regarding the information sources, and the topics of the required press review.

Interface agents are personal, not adaptive, and not cooperative (shortly $P\overline{AC}$). Personalization is supported to allow each user the customization of her/his interface. In the current implementation, adaptation is currently not supported, but -in general- an interface agent might adapt to the changes that occur in the preferences and interests of the corresponding user. Cooperation is not supported by agents belonging to this architectural level.

Table 6.2 summarizes the involved agents and their capabilities.

6.2.4 Experiments and Results

To evaluate the effectiveness of the system, several tests have been conducted using articles belonging to online newspapers [9].

Training Task Agents

First of all, several experiments have been performed to set the optimal parameters for the training activity. Through a suitable graphical interface (see Figure 6.4), the user can interact with the interface agents and sets her/him

[9] In the current implementation, www.repubblica.it, www.corriere.it, and www.espressonline.it

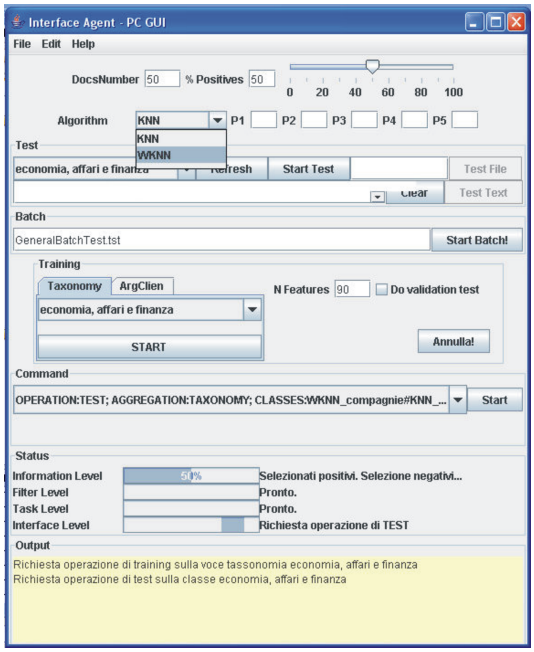


Figure 6.4: Interface for the newspaper articles classifying system.

preferences. In particular, experiments have been conducted adjusting the following parameters:

- the classification algorithm [10];
- the number of documents forming the dataset;
- the training category;
- the percentage of positive examples;
- the number of features to be considered.

First, task agents have been trained by a set of newspaper articles previously classified by experts of the domain. For each item of the taxonomy, a set of 200 documents has been selected to train the corresponding classifier.

Subsequently, to validate the training procedure of the first step of classification, the system has been fed by the same dataset used in the training phase, showing an accuracy between 96% and 100%. It is worth pointing out that, in this specific task, the accuracy should not be directly considered as a measure of the system performance. On the other hand, it becomes important since the accuracy of a classifier (evaluated on a balanced test set, i.e., with a number of negative examples that does not differ much from the

[10] In the current implementation kNN or WkNN

Table 6.2: Agents for text categorization.

Level	**Activity**	**Agents**	**Capabilities**
Information	web data extraction	*RSSWrappers*	$\overline{PAC}$
		HTMLWrappers	$\overline{P}A\overline{C}$
	wrapping sources	*IPTCWrapper*	$\overline{PAC}$
Filter	information filtering	*StopWordsRemover*	$\overline{PA}C$
	(preprocessing)	*Stemmer*	$\overline{PAC}$
		FeatureSelector	$PA\overline{C}$
Task	information processing	*Training*	PAC
	(text categorization)	*Test*	PAC
		FeedBack	PAC
Interface	input handling and	*PCInterface*	$P\overline{AC}$
	results presentation		$P\overline{AC}$
Middle	handling interations	Brokers:	$\overline{PA}C$

number of positive ones) indirectly affects the *recall*, under the hypothesis that classifiers are (dynamically) combined using logical operators and/or (statically) combined according to the given taxonomy (in this latter case, they are in fact in a pipeline).

Testing

To test the performance of the system, random datasets for each category have been generated. They were taken from a database containing articles previously classified by experts of the domain.

User choices are sent from the interface agent to the task level through the cooperation of the middle agent that belongs to the *task-interface* middle level (TI agent). The TI agent generates a task agent that embodies the corresponding classifier algorithm and asks it to perform the classification with the user preferences. The dataset needed for the classification is provided by information agents and subsequently pruned by the filter agents. After the classification activity, the task agent saves its own state in a suitable *XML*-like format in order to make it available for the test phase.

The accuracy for fourteen categories is summarized in Figure 6.5. On the average, the accuracy of the system is 80.05%. Particular care has been taken in limiting the phenomenon of "false negatives" (FN), which –nevertheless– had a limited impact on the percent of "false positives" (FP). In particular, the ratio $FN/(FN + FP)$ has been kept under 25% by weighting positive prototypes with an additional factor of 1.05 with respect to negative ones.

Results are very encouraging and show that the proposed approach is effective in the given application task, also taking into account that the system can be improved in several and important aspects.

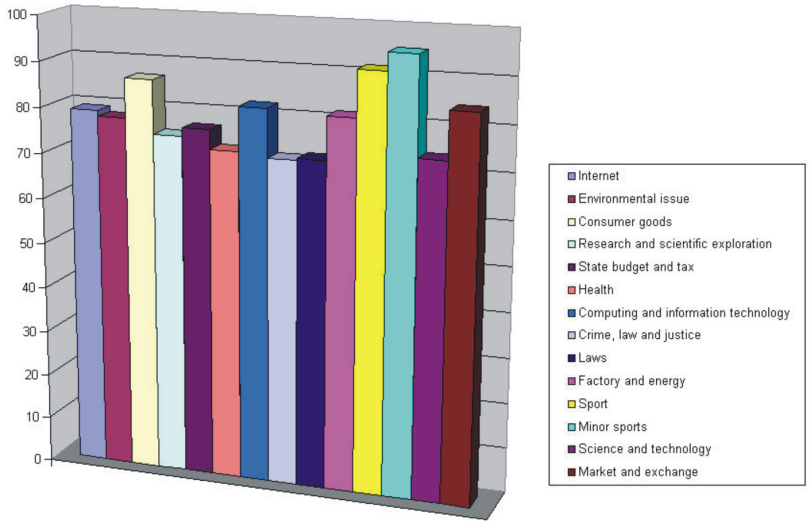


Figure 6.5: Accuracy of the system.

6.3 Summary

In this chapter, the peculiarities of the PACMAS architecture have been highlighted by depicting two relevant case studies.

The first one is the prototype of an e-service devoted to support undergraduate and graduate students in their university activities. It is able to retrieve relevant information from heterogeneous sources (e.g.: files, forums, databases, professor homepages, department web pages, etc.), and then filter, organize, and present it to the user, according to her/his personal needs and preferences.

The second one is a classification system devoted to create personalized press-reviews from online newspapers. It is able to extract from web sites of online newspapers the articles deemed relevant for a specific user, and select the relevant ones – according to specific user preferences – through suitable classifying algorithms. The categorization capability has been evaluated using several newspaper articles previously classified by hand by domain experts. Preliminary results are encouraging, showin an average global accuracy of about 80%.

Part III

Planning by Abstraction

Chapter 7

The Hierarchical Wrapper *HW[ ]*

This chapter presents a novel approach for implementing the planning capabilities (that is, the pro-active behaviour) of an intelligent agent.

So far, existing planning systems have been focusing on improving a specific algorithm (see section 3.5 for a survey), without taking into account the possibility of generalizing the approach to a generic algorithm.

This work concentrates on the possibility of exploiting abstraction techniques (in particular, organizing a planning domain into a hierarchy of abstract levels, i.e. an abstraction hierarchy) to improve the performance of a *generic* planner [ACV03c], instead of optimizing a particular planning algorithm. To this end, the parametric system *HW[ ]* has been devised and implemented to perform planning by abstraction (see [ACV03e] and [ACV03f]). The parameter is an external PDDL-compliant planner, which is exploited to search for solutions at any required level of the abstraction hierarchy, including the ground one.

Section 7.1 illustrates the architecture of the overall planner obtained by embedding a generic planner into the hierarchical wrapper *HW[ ]*. Next, section 7.2 describes the planning algorithm of the resulting hierarchical planner. Then, the extension to the standard PDDL notation that has been devised to represent abstraction hierarchies, is thoroughly illustrated in section 7.3. Finally, section 7.4 shows experiments made by embedding several different planners into the *HW[ ]* system.

7.1 System Architecture

HW[ ] stands for (parametric) Hierarchical Wrapper. Note that square brackets are part of the name, pointing to the ability of embedding an external planner. Being P any such planner, the notation *HW[P]* shall be used to denote an instance of *HW[ ]* able to exploit the planning capabilities of

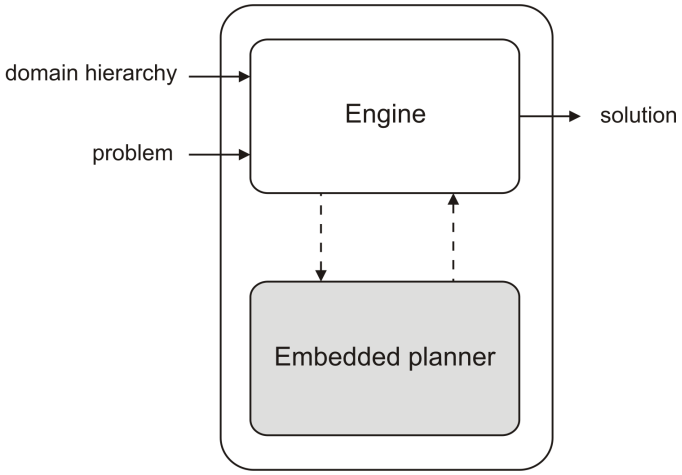


Figure 7.1: The *HW[ ]* Architecture.

P. The system can embed any domain-independent planner, provided that a compliance with the STRIPS subset of the PDDL1.2 standard is ensured. In principle, each level of abstraction may contain a different planner, thus permitting to select the most suitable planner for each level. In this case, a natural notation for highlighting the ordered set of embedded planners would be $HW[P_0, P_1, \ldots P_n]$, P_i being the planner embedded at the ith level of the hierarchy. In the following, we assume that the same external planner is used at each level of abstraction, and only one abstract level exists giving rise to a two-levels (i.e., ground and abstract) hierarchical description.

Figure 7.1 sketches the architecture of the system, focusing on its two main components, i.e., an engine and the embedded planner. The former controls the communication between adjacent levels, whereas the latter performs planning at any given level of abstraction.

7.2 The Planning Algorithm

Once instantiated with an external planner *P*, *HW[P]* takes as inputs a ground-level problem and a structured description of the corresponding domain, including a set of rules to be used while mapping ground into abstract states and vice-versa. In fact, to perform planning at different levels of abstraction, the engine of *HW[ ]* must operate bi-directional translations (upwards and downwards) to permit communication between adjacent levels. To find a solution of a given problem, first the engine of *HW[P]* translates the init and goal sections from the ground to the abstract level. *P* is then invoked to search for an abstract solution. Subsequently, each abstract operator is refined by repeatedly invoking *P*. The refinement of an abstract operator is performed by activating *P*, at the ground level, on the goal ob-

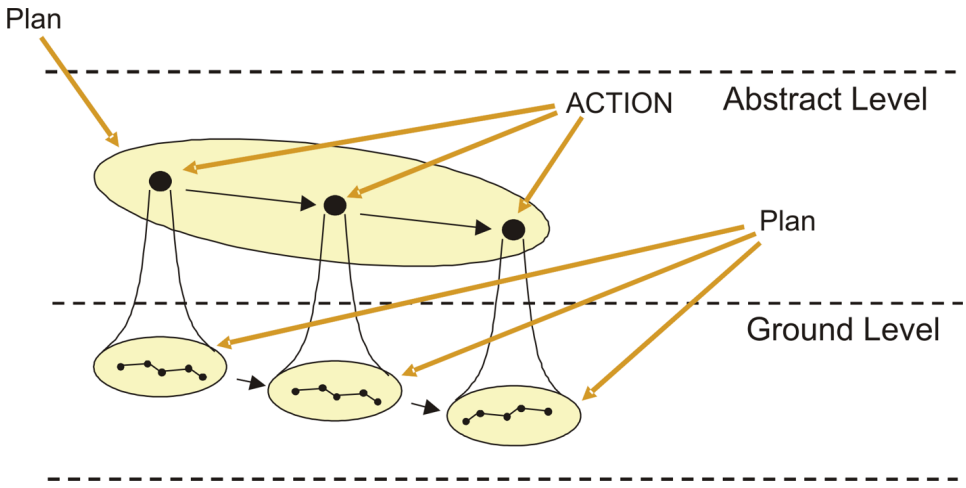


Figure 7.2: HW plan refinement process.

tained by translating downward its effects. Figure 7.2 shows the refinement process. Note that the initial state of each refinement depends on the previous refinement; hence, refinements must be performed according to the order specified by the abstract plan. To avoid incidental deletion of subgoals already attained during previous refinements, they are added to the list of subgoals that results from translating downward the effects of the current abstract operator to be refined. When the attempt to refine the current abstract solution fails, P is invoked to find the next abstract solution unless the number of abstract solutions found so far exceeds a given threshold. Note that, due to the limitations of most of the existing planners, the process of incrementally querying for another solution may be simulated by preliminarily querying for m abstract solutions to be released incrementally on demand. If no abstract solution could be successfully refined, to ensure the completeness of the algorithm an overall search is performed at the ground level. The whole process ends when a ground solution is found or the overall search fails.

7.3 An Extension to PDDL for dealing with Abstraction

Historically, several planning systems used abstraction hierarchies, e.g.: GPS (Newell and Simon 1972), ABSTRIPS (Sacerdoti 1974), ABTWEAK (Yang and Tenenberg 1990), PABLO (Christensen 1991), PRODIGY (Carbonell, Knoblock, and Minton 1990), but each of them introduced and adopted its own notation without following any standard. In other words, existing planning systems tailored for abstraction did not take into account the possibility

of introducing a common notation. To contrast the lack of a standard notation for supporting abstraction hierarchies, in this subsection a suitable extension to PDDL 1.2 is proposed [ACV03b].

As defined in section 4.3, an abstraction hierarchy consists of a list of formal systems $(\Sigma_0, \Sigma_1, \ldots, \Sigma_{n-1})$ with languages $\Lambda_0, \Lambda_1, \ldots, \Lambda_{n-1}$ respectively, and a list of effective total functions

$$f_k : \Lambda_k \rightarrow \Lambda_{k+1},$$

$$(k = 0, 1, \ldots, n-2)$$

devised to perform the mapping between adjacent levels of the hierarchy.

Assuming that standard PDDL is used to represent each $\Lambda_k (k = 0, 1, \ldots, n-1)$, in this section we focus on the problem of extending the standard for dealing with abstraction hierarchies, with particular emphasis on the mapping functions.

A problem and its corresponding domain are described in accordance with the standard PDDL 1.2 syntax, using the `define problem` and `define domain` statements, respectively.

The syntactic notation of the proposed extension is given according to the Extended BNF (EBNF), whose basics are briefly recalled, to avoid ambiguities:

- each production rule has the form `<syntactic element> ::= expansion`;
- angle brackets delimit names of syntactic elements;
- square brackets surround optional material;
- an asterisk means "zero or more of";
- a plus means "one or more of".

Furthermore, let us point out that here ordinary parentheses are an essential part of the grammar we are defining and do not belong to the EBNF meta language. To represent an abstraction hierarchy, the syntactic construct `define hierarchy` has been introduced, able to highlight the domains involved in the definition and the mapping between adjacent levels. It encapsulates an ordered set of domains, together with a corresponding set of mappings between adjacent levels of abstraction. Since the mappings are given in term of types, predicates and operators, three subfields have been defined (i.e. `:types`, `:predicates`, and `:actions`), to represent the abstraction over such dimensions. The general form of the construct is:

```
<hierarchy> ::=
(define (hierarchy <name>)
   <domain-def>
   (<mapping-def>*))
```

where:

```
<domain-def>::=
(:domains <domain name>+)
```

and:

```
<mapping-def>::=
  (:mapping (<source-domain> <destination-domain>)
    [:types <types-def>]
    [:predicates <predicates-def>]
    [:actions <actions-def>]
    [:invariants <invariants-def>])

<source domain> = <name>

<destination domain> = <name>

<types-def> ::= (<types-pair>+)

<types-pair> ::=
  (<destination type> <source type>)
<types-pair> ::= (nil <source type>)

<source type> = <name>

<destination type> = <name>

<predicates-def> ::= (<predicates-pair>+)

<predicates-pair> ::= (<predicate>  <PT>)
<predicates-pair> ::= (nil <PT>)

<predicate> ::=
  (<predicate name> <variable>*)

<variable> ::= ?<name>

<PT> ::= <typed-predicate>
<PT> ::= (and <PT>+)
<PT> ::= (or <PT>+)

<typed-predicate> ::=
(<predicate name> <typed list>*)
```

```
<typed list> ::=  <variable>+ - <type name>

<actions-def> ::= (<action-spec>+)

<action-spec> ::=
<action-pair> | <action-def>

<action-pair> ::= (<action> <AT>)
<action-pair> ::= (nil <AT>)

<action> ::= (<action name> <variable>*)
<AT> ::= <action>
<AT> ::= (and <AT>+)
<AT> ::= (or <AT>+)

<action-def> ::=
  see the PDDL 1.2 standard definition
```

Let us briefly comment the main definitions that occur within the proposed extension to PDDL, focusing on the underlying semantics.

7.3.1 Hierarchy Definition

As specified by the syntax, the `define hierarchy` statement contains two subsections: `<domain-def>` and `<mapping-def>`. The `:domains` field lists domains' names according to their abstraction level, from ground to the most abstract one. The `<mapping-def>` definitions specify the mapping between adjacent levels. In general, n levels of abstraction require $n - 1$ `<mapping-def>` definitions. Therefore, a single-level hierarchy would result in omitting the `<mapping-def>` definition (i.e., in this case only the ground level exists). It is worth noting that, although it would be desirable for the sake of clarity to give `:domains` and `:mapping` definitions (including `:types`, `:predicates`, and `:actions`) according to the ordering specified by the given grammar, nothing prevents from following a different ordering.

7.3.2 Mapping Definition

The `:mapping` field specifies the name of the source and destination domains, respectively. Given a source domain, the destination domain is unambiguously determined by consulting the `:domains` field. Nevertheless, for the sake of readability, the destination domain must be explicitly specified.

Types Definition

To represent how types are mapped between adjacent levels, in the `:types` field a list of clauses in the following notation must be given:

```
(<destination type> <source type>)
```

It specifies that `<source type>` becomes `<destination type>` while performing "upward" translations. In particular, `<source type>` is disregarded when the first argument of the pair equals to `nil`. For example, to disregard a *ground-type*, the following notation must be used:

```
(nil ground-type)
```

By default, if a type is not mentioned in any pair, it is forwarded unaltered to the destination level. If no `:types` field is provided, all constants and variables are forwarded to the destination level, labelling them with their `<source type>`.

Predicates Definition

The `:predicates` field declares how predicates are mapped between adjacent levels. Each `<predicates-pair>` expresses whether a predicate or a combination of predicates, obtained using logical `and`, `or`, and `not` operators, will be forwarded to the destination level. Generally speaking, three cases may arise:

- a predicate is forwarded unchanged: the pair can be omitted, being the default;
- a predicate is disregarded: the first argument becomes `nil`;
- a predicate is a logical combination of some predicates belonging to the source level: the second argument expresses the logical formula.

Note that the destination predicate accepts a list of untyped parameters, as in this case parameter types can be deducted from the `:types` mapping section. On the other hand, the source predicate needs to know the type of each parameter. This is required to avoid ambiguities, since there might be predicates with identical names, but different parameter types. If the `:predicates` field is entirely omitted, then no predicate-based abstraction occurs. In other words, each predicate is forwarded without any change to the upper level.

To map a predicate between adjacent levels, in the `:predicates` field the following notation must be used:

```
((abstract-predicate ?p11 ?p21 ...)
 (ground-predicate   ?p12  t12 ?p22  t22 ...))
```

It specifies that the ground-predicate must be preserved while going upward and vice-versa. If no differences exist in mapping a predicate between adjacent levels the corresponding clause can be omitted. To disregard a predicate while performing upward translations, the following notation is used:

```
(nil (ground-predicate  ?p12  t12 ?p22  t22 ...))
```

It specifies that ground-predicate is not translated into any abstract-level predicate. In addition, abstract-predicate can be expressed as a logical combination of some ground level predicates.

Actions Definition

To describe how to build the set of operators for the destination domain, in the `:actions` field four kind of mapping can be expressed:

1. An action is removed: the first argument becomes `nil`;
2. An action is expressed as a combination of actions belonging to the source domain (*parallelization* is expressed by `or` operator, whereas *serialization* is expressed by `and` operator);
3. An action remains unchanged or some of its parameters are disregarded: the pair can be omitted by default;
4. A new operator is defined from scratch: the statement `<action-def>` is used (note that this definition is not expanded in the notation, since it follows the standard PDDL 1.2).

Invariants Definition

As briefly pointed out in section 4.5, state invariants are fundamental to deal with the variable-unification process in operators, in order to avoid the problem of semantic inconsistencies.

To represent state invariants, one `:invariants` statement for each mapping definition between two adjacent levels should be added. In fact, in a *n-level* abstraction hierarchy, each mapping involves a specific set of invariants. Three kinds of invariants (identity, state membership, uniqueness of state membership) are supported. The general form of the `<invariants-def>` is the following:

```
<invariants-def>::=
  ([:identity <identity-def>]
   [:statemembership <statemembership-def>]
   [:uniqueness <uniqueness-def>])
```

```
<identity-def>::=
  (and <typed-predicate> <typed-predicate>+)
    ((= <variable> <variable>)+)

<statemembership-def>::=
  (or <typed-predicate> <type-predicate>+)

<uniqueness-def>::=
  (not (and <typed-predicate>
            <typed-predicate>+))
```

7.3.3 Examples of the Extension

In order to make clearer the proposed notation, in the following, some examples applied to a set of benchmark domains, are described.

Depots

Let us consider the *depots* domain, taken from the AIPS 2002 planning competition (Long 2002). The domain was devised by joining two well-known planning domains: *logistics* and *blocks-world*. They have been combined to form a domain in which trucks can transport crates around, to be stacked onto pallets at their destinations. The stacking is achieved using hoists, so that the resulting stacking problem is very similar to a blocks-world problem with hands. Trucks behave like "tables", since the pallets on which crates are stacked are limited. Let us suppose we want to create a two-level abstraction for the *depot* domain, composed by *depot-ground* and *depot-abstract*. According to the above notation, we can start defining the hierarchy in the following way:

```
(define (hierarchy depot)
  (:domains depot-ground depot-abstract)
  ...
```

Since there are only two levels of abstraction, just one `:mapping` statement is needed. To express the mapping rules (on types, predicates, and operators) from the ground to the abstract level, the following statement must be introduced:

```
  (:mapping
    (depot-ground depot-abstract)
    ...
```

Let us start with abstracting types of the *depot* domain type hierarchy (as reported in 7.3). We decided to disregard hoists and trucks, and not to

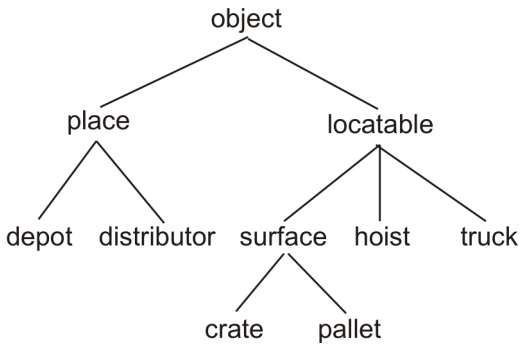


Figure 7.3: Type hierarchy for the *depots-ground* domain.

distinguish between depots and distributors (i.e., considering both as generic places).

According to the proposed notation, the translation can be expressed in the following way:

```
:types
  ((place depot)
   (place distributor)
   (nil hoist)
   (nil truck))
```

The first two statements assert that both `depot` and `distributor` become `place` in the *depot-abstract* domain. The last two statements assert that both `hoist` and `truck` must be disregarded. Let us recall that, by default, the types not mentioned remain unchanged at the abstract level (e.g. `locatable`, `crate`, `place`, etc.). The above notation entails the type hierarchy reported in 7.4.

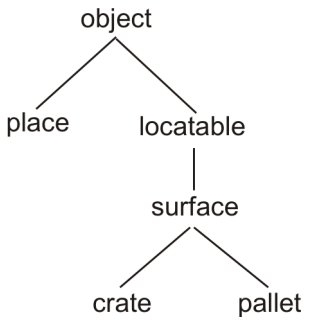


Figure 7.4: Type hierarchy for the *depots-abstract* domain.

The choice of removing some types implies that some predicates might become meaningless at the abstract level. In particular, predicates accepting parameters of type `truck` or `hoist` cannot exist at the abstract level.

```
(in ?c - crate ?t - truck)
(lifting ?h - hoist ?c - crate)
(available ?h - hoist)
(clear ?s - surface)
(on ?c - crate ?s - surface)
(at ?l - locatable ?p - place)
```

Figure 7.5: Predicates of the *depots-ground* domain.

7.5 lists the ground predicates of the *depot* domain. Since the `in` predicate accepts a `truck` as a parameter, it must be explicitly disregarded by the following statement:

```
(nil (in ?c  crate ?t  truck))
```

Similar considerations can be made for the `lifting` and `available` predicates. The predicates `(clear ?s  surface)` and `(on ?c  crate ?s  surface)` remain unchanged and can be omitted in the `:mapping` field (being the default). Note that `(at ?l  locatable ?p  place)` is overloaded, in the sense that it actually represents different predicates. Some examples of possible expansions are:

```
(at ?l  hoist ?p  distributor)
    (at ?l  truck ?p  depot)
    (at ?l  crate ?p  depot)
```

All expansions that accept any parameter whose type has been disregarded at the abstract level, must be explicitly removed. In this case, the following statements must be asserted:

```
 (nil (at?h  hoist ?p - place))
      (nil (at?t  truck ?p - place))
```

Let us point out that more complex mapping rules are admissible. For example, two or more ground predicates could be combined to form a new abstract predicate. Let us consider the statement below:

```
((moveable ?c ?h ?s ?p)
 (and (lifting ?h  hoist ?c  crate)
      (at ?h  hoist ?p  place)
      (clear ?s  surface)
      (at ?s  surface ?p  place))
```

The new predicate `moveable` is introduced, which applies only when the specified group of ground predicates are true. The mapping rules enforced on types and predicates may modify preconditions and effects of some ground operators. For example, consider the `drive` action:

```
(:action drive
  :parameters
   (?t - truck ?p1 ?p2 - place)
  :precondition
   (and (at ?t ?p1))
  :effect
   (and (not (at ?t ?p1))(at ?t ?p2)))
```

Since the `(at ?t truck ?p place)` predicate has not been forwarded to the abstract level, the `drive` action could not require any such precondition or effect. Therefore, `drive` becomes meaningless at the abstract level, and must be removed throughout the following statement:

```
((nil (drive?t ?p1 ?p2))
```

Similar considerations can be made for the load and unload actions:

```
 (nil (load?h ?c ?t ?p))
 (nil (unload?h ?c ?t ?p))
```

At this point, one may want to join the remaining actions `lift` and `drop` to form a new abstract operator (say `lift-and-drop`). According to the proposed extension, the new operator is defined as:

```
    ((lift-and-drop ?c ?s1 ?s2 ?p1 ?p2)
     (and (lift?h ?c ?s1 ?p1)
          (drop ?h ?c ?s2 ?p2)))
```

Moreover, the `lift` and `drop` actions can be ignored:

```
       (nil (lift?h ?c ?s ?p))
       (nil (drop?h ?c ?s ?p))
```

Alternatively, the new abstract operator `lift-and-drop` could be introduced from scratch as follows:

```
(:action lift-and-drop
  :parameters
   (?c - crate ?s1 ?s2  surface
    ?p1 ?p2 - place)
  :precondition
    (and (at ?c ?p1) (on ?c ?s1)
         (clear ?c) (at ?s2 ?p2)
         (clear ?s2))
  :effect
    (and (not (at ?c ?p1))
         (at ?c ?p2)(clear ?s1)
         (not (clear ?s2))
         (on ?c ?s2)
         (not (on ?c ?s1))))
```

For the sake of completeness, the entire hierarchy definition for the *depot* domain is summarized in 7.6.

```
(define hierarchy depots)
  (:domains depots-ground depots-abstract)
  (:mapping (depots-ground depots-abstract)
     :types
       ((place depot)
        (place distributor)
        (nil hoist)
        (nil truck))
     :predicates
       ((nil (lifting ?h - hoist ?c - crate))
        (nil (available ?h - hoist))
        (nil (in ?c - crate ?t - truck))
        (nil (at ?h - hoist ?p - place))
        (nil (at ?t - truck ?p - place)))
     :actions
      ((nil (drive ?t ?p1 ?p2))
       (nil (load ?h ?c ?t ?p))
       (nil (unload ?h ?c ?t ?p))
       (nil (lift ?h ?c ?s ?p))
       (nil (drop ?h ?c ?s ?p))
       ((lift-and-drop ?c ?s1 ?s2 ?p1 ?p2)
         (and (lift ?h ?c ?s1 ?p1)
              (drop ?h ?c ?s2 ?p2))))))
```

Figure 7.6: Hierarchy definition for the *depots* domain.

Elevator

In the above example, we started by abstracting the type hierarchy. It is worth pointing out that this choice is not mandatory; in fact abstraction could also be started by specifying the mapping of predicates or actions. To better illustrate an alternative approach, let us consider another example applied to the *elevator* domain (Koehler and Schuster 2000), whose ground definition is reported in 7.7. The type hierarchy of *elevator* is very simple and contains only two types: `passenger` and `floor`. Thus, let us abstract the domain from predicates. In particular, one may decide to disregard `(above ?f1 ?f2 floor)` and `(lift-at ?f floor)`, so that the `lift` is always available and moveable from a floor to another. This choice has an influence on actions: `up` and `down` become meaningless, whereas preconditions and effects of `board` and `depart` undergo some modifications on their abstract

```
(define (domain elevator-ground)
 (:requirements :typing)
 (:types passenger floor)
 (:predicates
  (origin ?person  passenger ?floor - floor)
  (destin ?person - passenger ?floor - floor)
  (boarded ?person - passenger)
  (served ?person - passenger)
  (above ?f1 ?f2 - floor)
  (lift-at ?floor - floor))
 (:action board
  :parameters (?f - floor ?p - passenger)
   :precondition (and (lift-at ?f) (origin ?p ?f))
   :effect (and (boarded ?p)))
   [...]
 (:action down
  :parameters (?f1 ?f2 - floor)
   :precondition (and (lift-at ?f1) (above ?f2 ?f1))
   :effect (and (lift-at ?f2) (not (lift-at ?f1)))))
```

Figure 7.7: The *elevator* domain.

counterparts (say `load` and `unload`, respectively). 7.8 shows the described hierarchy definition for the elevator domain.

Blocks-world

As an example of abstraction starting from actions, let us consider the *blocks-world* domain, reported in 7.10. In this case the type hierarchy cannot be abstracted, as it contains only the type `block`. In this domain two macro-operators can be identified: *pick-up;stack* and *unstack;put-down*. The decision of adopting these operators entails a deterministic choice on which predicates have to be forwarded / disregarded while performing upward translations. More explicitly `(handempty)` and `(holding ?b  block)` must be disregarded, meaning that the "hand" can be considered always available. 7.11 shows the corresponding hierarchical definition of the *blocks-world* domain, according to the proposed notation.

7.4 Experiments and Results

To experiment with the proposed abstraction mechanism, a prototype of the system has been implemented in C++. Experiments have been performed with three planners: *GRAPHPLAN* [BF95], *BLACKBOX* [KS98],

```
(define (hierarchy elevator)
  (:domains elevator-ground elevator-abstract)
  (:mapping
     (elevator-ground elevator-abstract)
     :predicates
       ((nil (lift-at?f  floor))
        (nil (above ?f1 ?f2 - floor)))
      :actions
       ((nil (up?f1 ?f2))
        (nil (down?f1 ?f2))
        (nil (board?f ?p))
        (nil (depart?f ?p))
        ((load ?f ?p) (board ?f ?p))
        ((unload ?f ?p) (depart ?f ?p)))))
```

Figure 7.8: Hierarchy definition for the *elevator* domain.

```
(define (domain elevator-abstract)
 (:requirements :typing)
 (:types passenger floor)
 (:predicates
  (origin ?person - passenger ?floor - floor)
  (destin ?person - passenger ?floor - floor)
  (boarded ?person - passenger)
  (served ?person - passenger))
 (:action load
  :parameters (?p - passenger ?f - floor)
  :precondition (and (origin ?p ?f))
  :effect (and (boarded ?p)))
 (:action unload
  :parameters (?p - passenger ?f - floor)
  :precondition (and (boarded ?p) (destin ?p ?f))
  :effect (and (served ?p))))
```

Figure 7.9: The *elevator-abstract* domain.

```
(define (domain blocks)
 (:requirements :typing)
 (:types block)
 (:predicates
   (on ?x ?y - block) (ontable ?x - block) (clear ?x - block)
   (handempty) (holding ?x - block))
 (:action pick-up
   :parameters (?x - block)
   :precondition (and (clear ?x) (ontable ?x) (handempty))
   :effect (and (not (ontable ?x)) (not (clear ?x))
                (not (handempty)) (holding ?x)))
 (:action put-down
   :parameters (?x - block)
   :precondition (holding ?x)
   :effect (and (not (holding ?x)) (clear ?x) (handempty) (ontable ?x)))
 (:action stack
   :parameters (?x ?y - block)
   :precondition (and (holding ?x) (clear ?y))
   :effect (and (not (holding ?x)) (not (clear ?y)) (clear ?x)
                (handempty) (on ?x ?y)))
  (:action unstack
    :parameters (?x ?y - block)
    :precondition (and (on ?x ?y) (clear ?x) (handempty))
    :effect (and (holding ?x) (clear ?y) (not (clear ?x)) (not (handempty))
             (not (on ?x ?y)))))
```

Figure 7.10: The *blocks-world* domain.

```
(define (hierarchy blocks)
  (:domains blocks-ground blocks-abstract)
  (:mapping
    (blocks-ground blocks-abstract)
    :predicates
      ((nil handempty))
       (nil (holding ?b - block)))
    :actions
      ((nil (pick-up ?b))
       (nil (put-down ?b))
       (nil (stack ?b1 ?b2))
       (nil (unstack ?b1 ?b2))
       ((pick-up&stack ?b1 ?b2)
          (and (pick-up ?b1) (stack ?b1 ?b2)))
       ((unstack&put-down ?b1 ?b2)
          (and (unstack ?b1 ?b2)
               (put-down ?b1))))))
```

Figure 7.11: Hierarchy definition for the *blocks-world* domain.

and *LPG* [GS02]. In the following, *GP*, *BB*, and *LPG* shall be used to denote the *GRAPHPLAN*, *BLACKBOX*, and *LPG* algorithms, whereas *HW[GP]*, *HW[BB]*, and *HW[LPG]* shall be used to denote their hierarchical counterparts. To assess the capability of the proposed approach to improve the performance of the search, some tests on five domains taken from the 1998, 2000, and 2002 AIPS planning competitions ([Lon98], [Bac00], [Lon02]) have been performed.

In particular, the domain chosen for the experiments are: *elevator*, *logistics*, *blocks-world*, *zeno-travel*, and *gripper*. Experiments were conducted on a machine powered by an Intel Celeron CPU, working at 1200 Mhz and equipped with 256Mb of RAM. A time bound of 1000 CPU seconds has also been adopted, and the threshold (m) used to limit the search for abstract solutions has been set to 1 for each planner. All domains have been structured according to a ground and an abstract level; for each domain, several tests have been performed characterized by increasing complexity. 7.4 compares the CPU time of each planner over the set of problems taken from the AIPS planning competitions. Dashes show problem instances that could not be solved by the corresponding system within the adopted time-bound.

7.4.1 Elevator

The elevator domain was designed to optimize the travel routes of lifts through a new intelligent lift controller based on AI Planning techniques.

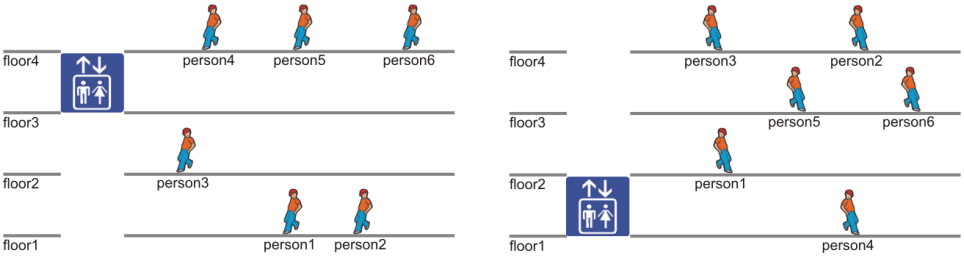


Figure 7.12: Example of problem for the *elevator* domain.

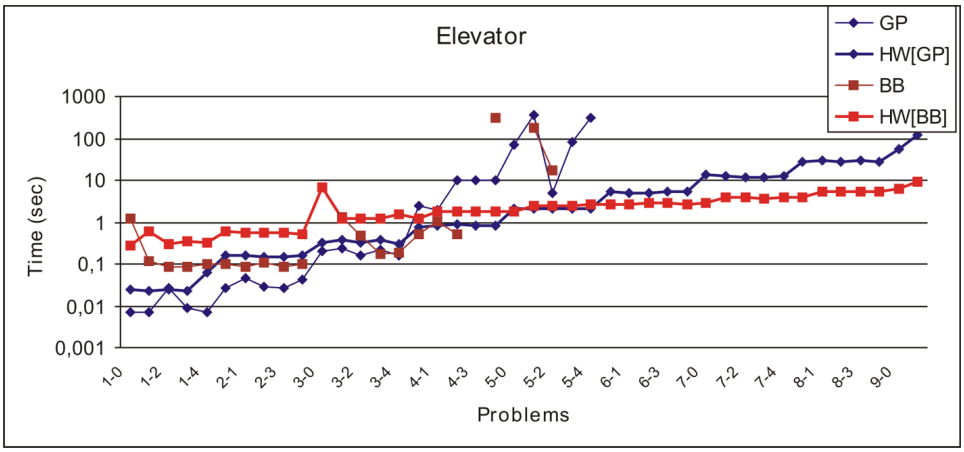


Figure 7.13: Results for the *elevator* domain.

The domain is formed by a lift, which must take passengers from one floor to another. For each passenger the source and destination floors are known in advance. The goal is to serve all passengers. Figure 7.12 shows an example of initial and goal state in this domain. Experiments confirm that the complexity of the domain increases rapidly with the number of passengers and floors. In fact, for *GP* and *BB* the CPU time increases very rapidly while trying to solve problems of increasing length, whereas *HW[GP]* and *HW[BB]* keep solving problems with greater regularity (although the relation between number of steps and CPU time remains exponential). *LPG* is able to solve long plans in a very short time, thus doing away the need to resort to *HW[LPG]*. Figure 7.13 summarizes results obtained in the elevator domain. Note that Y axis is expressed in logarithmic scale.

7.4.2 Logistics

The logistics domain involves transportation of packages among cities. Cities contain several locations, some of which are airports. Trucks move packages among locations and/or airports. Planes move packages between two airports. The complexity of the domain grows with the number of objects

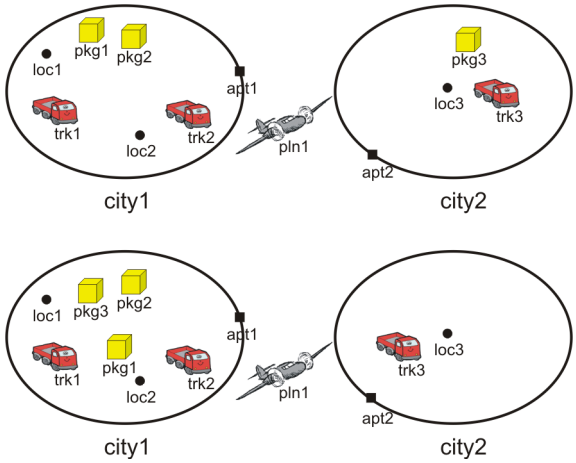


Figure 7.14: Example of problem for the *logistics* domain.

(packages, trucks, planes, cities) involved. Figure 7.14 shows an example of a problem in the logistics domain. In this domain *GP* easily solves problems up to a certain length but it is unable to solve problems within the imposed time limits if a given threshold is exceeded. On the other hand, *HW[GP]* keeps solving problems of increasing length without encountering the above difficulties. *BB* performs better than *HW[BB]* for small problems, whereas *HW[BB]* outperforms *BB* on more complex problems. *LPG* is able to solve long plans in a few seconds at the most. For unknown reasons *LPG* was not able to refine any abstract operator when invoked by the engine of *HW*. Figure 7.15 summarizes the results.

7.4.3 Blocks-world

The blocks world is one of the oldest domain known in the artificial intelligence community. It has been widely used to test a number of planning algorithms, since it is only apparently simple. For this reason it has been chosen as test-bed to experiment abstraction techniques devised in this work. The domain is formed by a finite set of blocks and a table large enough to accommodate all the blocks. Each block is either on another block or on the table. No block can be on two places at the same time (physical restriction), and the table is always clear. The actions allowed in the domain are: taking/putting a block from/onto a set of stacked blocks, pick up a block from the table, put down a block on the table. Figure 7.16 shows an example of problem in this domain.

As shown in 7.17, tests performed on the *Blocks-world* domain reveal a similar trend for *GP* and *HW[GP]*, although the latter performs slightly better than the former. *BB* performs better than *HW[BB]* for simple problems, whereas *HW[BB]* outperforms *BB* on problems of medium complexity. *LPG*

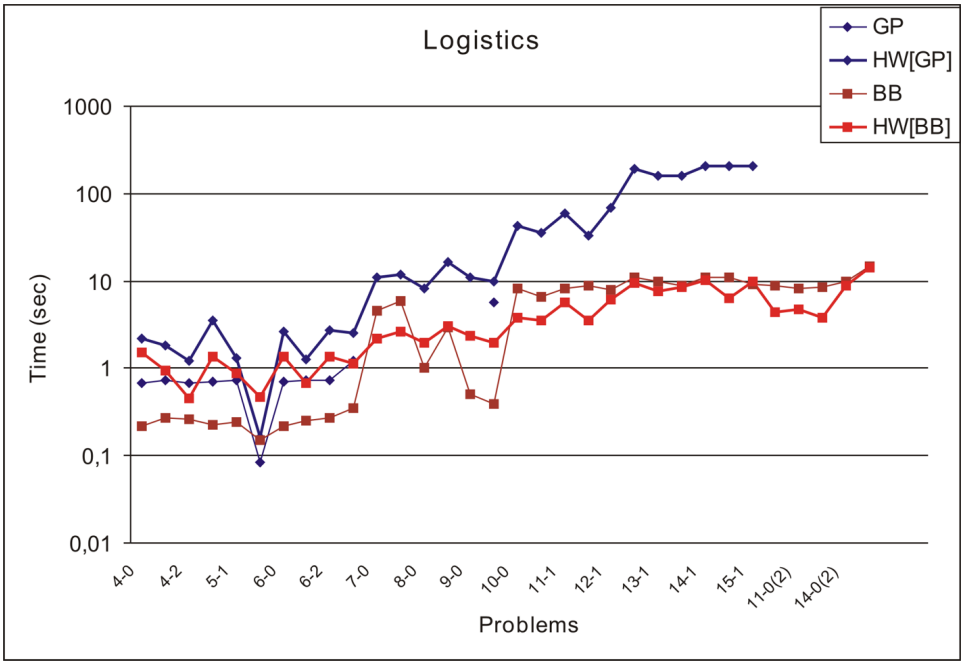


Figure 7.15: Results for the *logistics* domain.

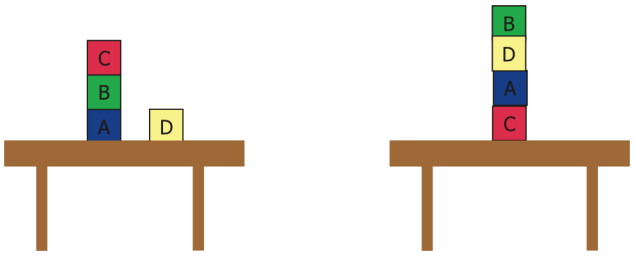


Figure 7.16: Example of problem for the *blocks-world* domain.

is able to solve problems whose solution length is limited to 100 steps. In this domain, *HW[LPG]* clearly outperforms *LPG* on more complex problems as shown in Figure 7.18.

7.4.4 Zeno-travel

The zeno-travel domain involves transporting people around in planes, using different modes of movement: fast and slow. The fast movement consumes fuel faster than slow movement, making the search for a good quality plan (one using less fuel) much harder. Figure 7.19 illustrates a problem in the zeno-travel domain. Unfortunately, neither *GP* nor *HW[GP]* are able to successfully tackle any problem of this domain. An improvement of *HW[BB]* over *BB* can be observed, similar to the one shown for the blocks-world

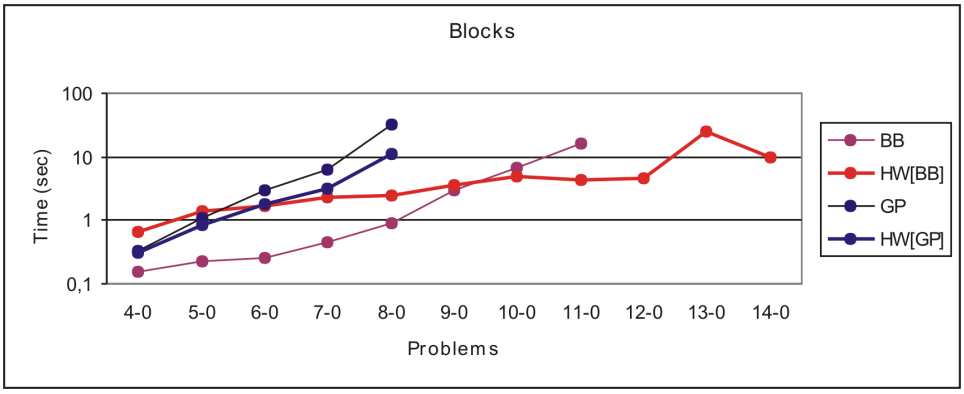


Figure 7.17: Perfomance comparison between GP, BB and their hierarchical counterparts in the *blocks-world* domain.

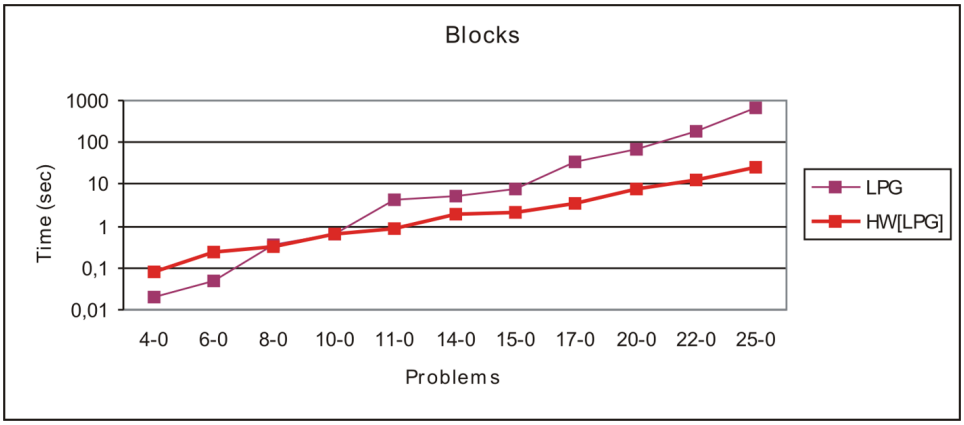


Figure 7.18: Perfomance comparison between LPG and HW[LPG] in the *blocks-world* domain.

domain. *LPG* is able to solve long plans in a few seconds at the most, thus avoiding the need to resort to *HW[LPG]*.

7.4.5 Gripper

Figure 7.20 shows a simple example of problem in the gripper domain. This domain involves the trasportation of balls from a room to another, by a robot equipped with two grippers. As shown in figure 7.21 , for the gripper domain, both *HW[GP]* and *HW[BB]* clearly outperform their non-hierarchical counterparts. *LPG* is able to solve long plans in a very short time.

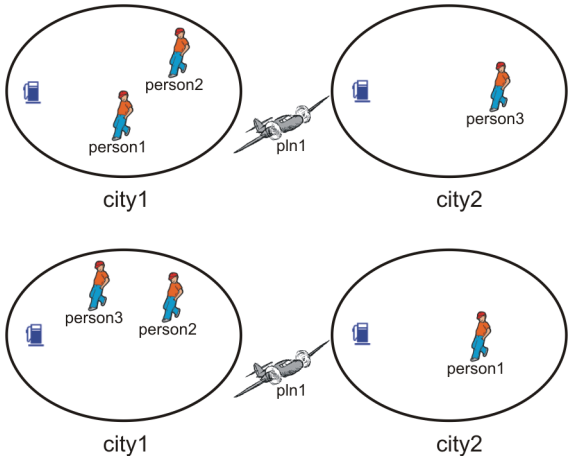


Figure 7.19: Example of problem for the *zeno-travel* domain.

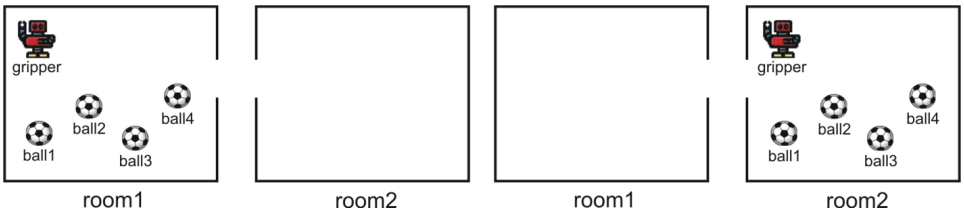


Figure 7.20: Example of problem for the *gripper* domain.

7.5 Summary

In this chapter, a novel parametric system has been presented, devised to perform planning by abstraction. The actual search is delegated to an external planners, that is the parameter. Aimed at giving a better insight of whether or not the exploitation of abstract spaces can be useful for solving complex planning problems, comparisons have been made between any instances of the hierarchical planner and its non-hierarchical counterpart. In order to handle abstraction hierarchies, a suitable extension to the PDDL standard notation, has been devised. To better investigate the significance of the results, three different planners have been used to make experiments. A set of problem of increasing complexity, taken from five significant benchmark domains, has been chosen to perform experiments. Experimental results highlight that abstraction is useful on classical planners, such as GP and BB. On the contrary, the usefulness of resorting to hierarchical planning for the latest-generation planner used for experiments (i.e., LPG) clearly emerges only in the blocks-world domain.

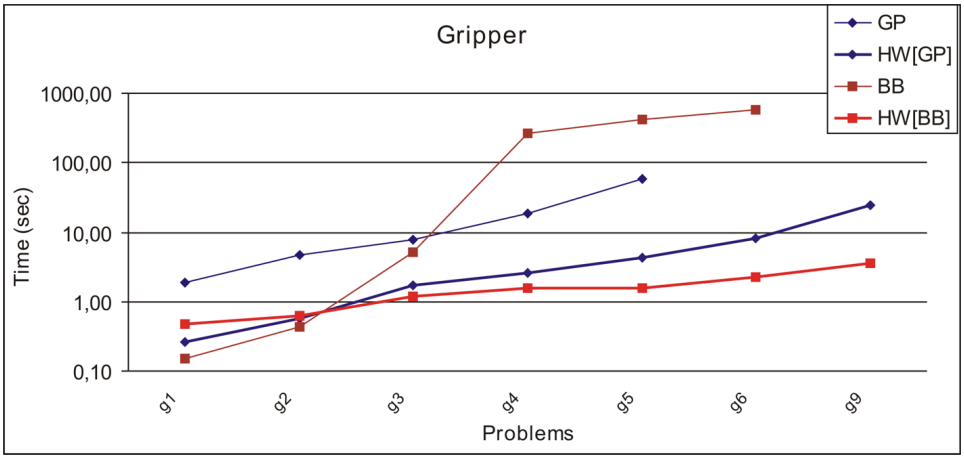


Figure 7.21: Results in the *gripper* domain.

#	GP	HW[GP]	BB	HW[BB]	LPG	HW[LPG]
elevator						
1-4	0.01	0.06	0.1	0.33	0.01	0.11
3-1	0.23	0.36	1.34	1.20	0.02	0.15
4-1	1.96	0.83	1.03	1.74	0.02	0.16
4-4	10.11	0.84	311.5	1.79	0.02	0.16
5-1	364.7	2.03	180.8	2.54	0.02	0.18
7-2	–	12.04	–	3.89	0.03	0.29
logistics						
4-2	0.68	1.22	0.27	0.46	17.93	–
5-2	0.08	0.16	0.15	0.46	0.02	–
7-0	–	10.93	4.49	2.17	2.12	–
8-1	–	16.26	2.90	3.02	1.55	–
10-0	–	43.43	8.27	3.76	2.17	–
15-0	–	203.4	10.91	6.33	0.15	–
blocks-world						
4-0	0.34	0.32	0.16	0.67	0.02	0.08
6-0	30.4	1.82	0.26	1.68	0.05	0.23
8-0	31.61	11.13	0.92	2.46	0.36	0.31
10-0	–	–	6.82	5.00	0.62	0.67
11-0	–	–	16.23	4.25	4.23	0.83
14-0	–	–	–	9.84	5.00	1.91
15-0	–	–	–	–	7.49	2.07
17-0	–	–	–	–	33.93	3.49
20-0	–	–	–	–	66.78	7.88
22-0	–	–	–	–	183.16	12.21
25-0	–	–	–	–	668.98	24.94
zeno-travel						
1	0.02	0.52	0.22	0.36	0.02	0.03
8	–	42.55	0.94	2.36	0.14	0.49
9	–	–	0.34	3.37	0.13	1.08
11	–	–	11.20	2.78	0.16	1.06
13	–	–	62.99	20.52	0.42	2.47
14	–	–	–	20.04	3.90	21.93
gripper						
2	4.72	0.56	0.42	0.63	0.02	0.07
3	7.91	1.73	5.22	1.20	0.02	0.12
4	18.32	2.63	268.7	1.55	0.02	0.14
5	57.21	4.38	421.1	1.54	0.03	0.15
6	–	7-97	586.4	2.26	0.03	0.17
9	–	24.29	–	3.63	0.05	0.36

Table 7.1: Performance comparison of *BB*, *GP*, and *LPG* together with their hierarchical counterparts *HW[BB]*, *HW[GP]*, *HW[LPG]*.

Chapter 8

The Domain Hierarchy Generator *DHG*

The previous chapter presented the *HW[ ]* system, a novel approach for implementing the planning capabilities of an intelligent agent. It uses abstraction techniques to improve the performance of the search in complex domains. The systems is able to exploit a planning domain organized into a hierarchy of abstract levels (i.e. an abstraction hierarchy) to improve the performance of a *generic* planner. However, it is long and often difficult to made by hand abstraction hierarchies.

This chapter focuses on the problem of automatically generating the abstraction hierarchies [ACV03d]. In particular, it addresses the problem of how to identify macro-operators starting from a ground-level description of a domain, to be used for generating useful abstract-level descriptions [ACV04b].

In the following, the DHG system [ACV05] – devised to automatically generate abstraction hierarchies – will be described.

8.1 Introduction

The attempt of dealing with the complexity of planning tasks by resorting to abstraction techniques is a central issue in the field of automated planning. Although the generality of the approach has not been proved always useful on domains selected for benchmarking purposes, it will play a central role as soon as the focus will move from artificial to real problems. Studying abstraction may have a great impact on the "internals" of intelligent agents, since -by definition- an agent must be able to generate plans in arbitrarily complex domains, no matter which environment, real (e.g., robotic applications) or virtual (e.g. Internet, computer games), is being considered. Therefore, it will be crucial to have a tool for automatically generating abstraction hierarchies from a domain description.

To this end, a system that -given a description of the domain expressed in PDDL- outputs a set of macro-operators to be used as a starting point for defining abstract operators, has been implemented.

8.2 The DHG system

As described in section 4.3, a planning domain can be defined in terms of two kinds of entity: *predicates* and *operators*. A particular kind of unary predicate can also be taken into account, giving rise to a third kind of entity, i.e. *types*, possibly organized according to a suitable "is-a" hierarchy. Although, in principle, abstraction might be performed on both predicates (including types) and operators, the work described in this section is mainly concerned with abstractions on operators. In particular, a novel approach for the automatic extraction of macro-operators is presented.

It is worth noting that the definition of abstract operators is strictly related with the definition of abstract predicates and vice versa. Keeping this in mind, this proposal can be positioned between action- and state-based techniques.

For the sake of simplicity, let us consider only two abstraction levels, namely *ground* and *abstract*. Note that ground and abstract domains have the same form and are loosely related under the assumption that (most of the) abstract plans should be refinable at the ground level. To guarantee this desirable property, an abstract operator should be defined on top of several (at least one) supporting macro-operators, i.e., macro-operators whose pre- and post-conditions match the one defined for the corresponding abstract operator. On the other hand, a macro-operator can be obtained by uninstantiating any legal sequence of ground operators.

To tackle a planning problem using abstraction, one (or more) abstract level(s) starting from the ground one should be defined. Abstracting a ground domain leads to the definition of an abstraction hierarchy, consisting of a set of predicates and operators, together with a mapping function devised to specify the mapping between ground and abstract level. In general, three kinds of mappings should be defined: (i) a set of types at the ground level can be represented by a single type at the abstract level, (ii) a single predicate at the ground level can be represented by a logical combination of predicates at the abstract level, and (iii) a set of macro-operators at the ground level can be combined into a single operator at the abstract level.

There is no predefined ordering in the abstraction process. In fact, one may start abstracting types, rather than predicates or operators although any choice performed on one kind of mapping may impact on subsequent choices. Nevertheless, as the section is mainly concerned on automatically extracting macro-operators, let us adhere to the underlying assumption that our concerns about predicates (and types) play a secondary role, with respect

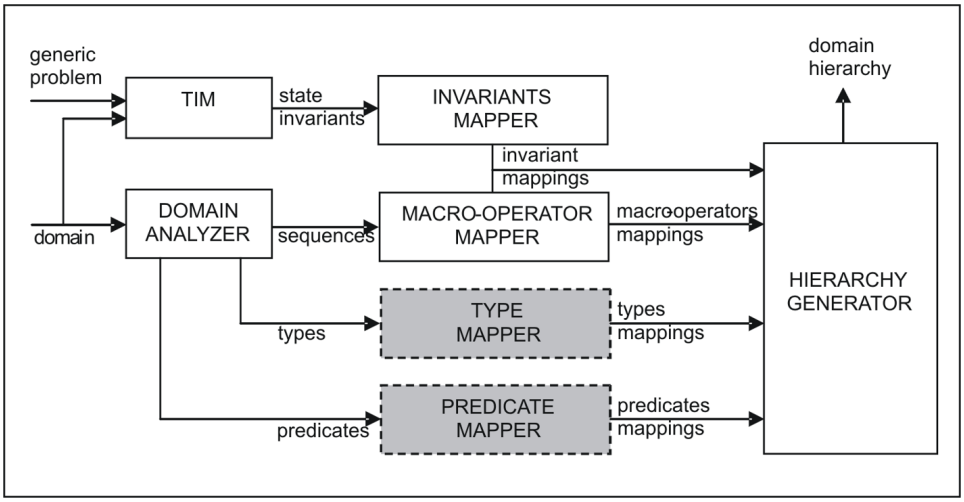


Figure 8.1: The *DHG* architecture.

to operators, in the process of defining an abstraction hierarchy.

Figure 8.1 depicts the architecture of the system –called *DHG*, standing for Domain-oriented Hierarchy Generator– devised to automatically generate the abstraction hierarchies.

The hierarchy generator module currently takes as inputs: (i) state invariants mappings (generated by the invariants mapper that processes the output produced by TIM [FL98]), and (ii) supporting macro-operators mappings (extracted from the sequences given by the DOMAIN ANALYZER described in the following). *DHG* outputs a domain hierarchy, consisting of a ground and an abstract level. Currently, abstract operators and predicates are generated according to a simple strategy: for each supporting macro-operator a different abstract operator is generated, whose pre- and post-conditions are made coincident with the selected macro-operator. All predicates not involved in any pre- or post-condition are deleted from the abstract domain.[1]

8.2.1 Generating Abstraction

Generating abstraction basically involves executing two steps: (i) searching for macro-operator schemata through a priori or a posteriori analysis, (ii) selecting some of the schemata evidenced so far and translating them into abstract operators.

[1]The final system, consisting of additional modules devised to map also types and predicates (shadowed in the figure), will be able to perform abstraction along all the cited dimensions -i.e., predicates, types, and operators.

As briefly said in section 4.5, macro-operators can be obtained by resorting to "a posteriori" or "a priori" analysis. In this section, we concentrate on the task of finding macro-operator schemata throughout an *a-priori* analysis performed on the given domain and problem, rather than adopting the *a-posteriori* technique

Searching for macro-operator schemata

The core of the whole process consists of finding a set of relevant sequences and then (possibly) promoting them to macro-operators. The basic steps for identifying such sequences are performed by the `DOMAIN ANALYZER` using a graph-oriented technique: first of all, a directed graph containing information about the dependencies between ground operators is built. Being G such graph, its nodes represent ground operators, and its edges represent relations between effects of the source node and preconditions of the destination node. In particular, for each source node A and for each destination node B, the corresponding edge is labelled with a pair of non-negative numbers, say $< a\ b >$. The pair accounts for how many predicates A can establish (a) and negate (b) that are also preconditions of B. Note that source and destination node may coincide, thus giving rise to a self-reference. From each acyclic path, a relevant sequence of operators can be extracted. As considering all possible paths would end up to a large amount of macro-operators, a second step consists of pruning G yielding the pruned graph G_p. The pruning activity is controlled by a set of domain-independent heuristics reported in table 8.1

Note that the pruned graph does not contain edges labeled $< 0\ 0 >$, the corresponding operators being completely independent. A set of sequences (candidates to generate macro-operators) is then extracted from G_p, each path being related with a relevant sequence. In particular, sequences whose post-conditions are represented by empty sets are disregarded for obvious reasons. The remaining sequences are considered for generating macro-operators.

As an example, let us consider the well-known *blocks-world* domain, encompassing four operators: `stack`, `pick-up`, `unstack`, `put-down`. The corresponding graph is shown in Figure 8.2.

Bearing in mind that the same mechanism has been applied to all operators pairs, let us concentrate for instance on the relation that holds between `stack` (source node) and `pick-up` (destination node). Considering that the effects of the stack operator are:

```
(not (holding ?x))
(not (clear ?y))
(clear ?x)
(handempty)
```

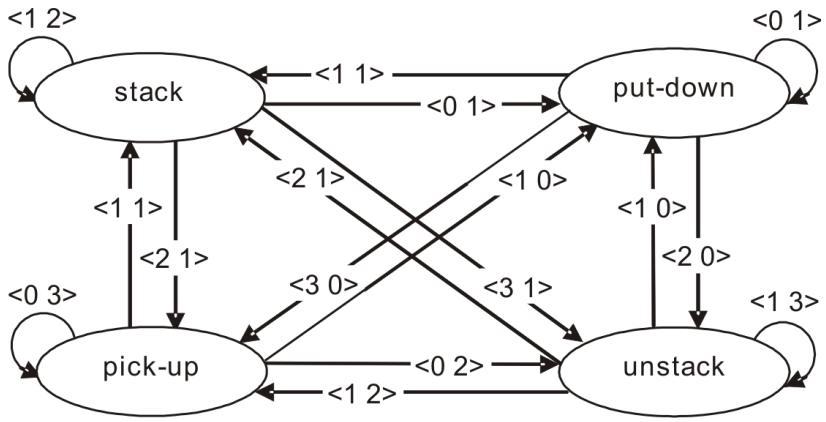


Figure 8.2: The directed graph (before pruning), representing static relations between operators of the *blocks-world* domain.

```
(on ?x ?y)
```

and that the preconditions of the `pick-up` operator are:

```
(clear ?x)
(ontable ?x)
(handempty)
```

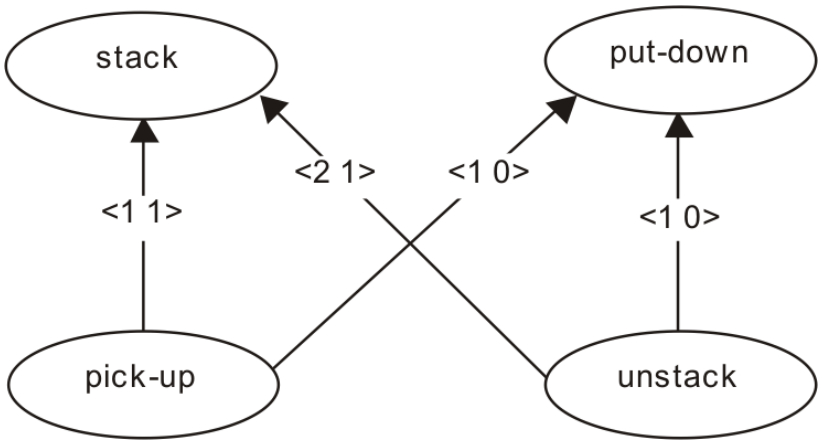


Figure 8.3: The directed graph (after pruning), representing static relations between operators of the *blocks-world* domain.

we label the corresponding edge with the pair <2 1>. It is apparent that `stack` establishes two preconditions for `pick-up`, while negating another. As for the pruning activity, figure 8.3 shows the resulting graph for the *blocks-*

world domain.[2] The resulting macro-operator schemata are (";" being used for concatenation): *pick-up;stack*, *unstack;put-down*, *pick-up;put-down*, and *stack;unstack*.

Translating relevant schemata into abstract operators

Step (ii) consists of extracting sequences from the pruned graph and then converting them into macro-operators. The relevant sequences can be easily extracted from the pruned graph, each path being related with a candidate macro-operator. Among all existing paths, only those containing a single occurrence of each operator are selected.

For each extracted relevant sequence, a corresponding macro-operator is generated, whose pre- and post-conditions are evaluated from pre- and post-conditions of the operators belonging to the sequence. Each extracted macro-operator is then promoted to an abstract operator, defined according to the define action statement of the standard PDDL notation by its name, together with its parameters, its pre- and post-conditions.

Let us formally represent the process of promoting a sequence of ground operators to a macro-operator. In particular, let us assume that σ is a sequence of operators , whose application to the source state S_1 leads to the destination state S_2. Under this assumption, a corresponding macro-operator can be defined as follows where γ, η, α, and δ represent preconditions, effects, add-list, and delete-list of the resulting macro-operator, respectively:

$$\begin{cases} \gamma_\sigma = \gamma_{\sigma_1} \cup (\gamma_{\omega_n} \setminus \eta_{\sigma_1}) \\ \alpha_\sigma = (\alpha_{\sigma_1} \setminus \delta_{\omega_n}) \cup (\alpha_{\omega_n} \setminus \gamma_{\sigma_1}) \\ \delta_\sigma = (\delta_{\omega_n} \setminus \alpha_{\sigma_1}) \cup (\delta_{\sigma_1} \setminus \alpha_{\omega_n}) \end{cases} \tag{8.1}$$

The formula 8.1 can be easily evaluated if all the actions belonging to σ are instantiated (i.e. all the involved parameters refer to a specified object). On the contrary, applying the formula in presence of variables could lead to semantic inconsistencies.

A typical example that highlights this problem occurs when predicates that account for spatial relations are considered. For instance, while considering the predicate `(at ?o - object ?l - location)`, used in the `Logistics` domain to represent the position of an object, there cannot be two predicates stating that the same object is in two different locations. This condition can be expressed through the use of suitable state invariants. These are not explicitly stated in the domain description and can be retrieved using TIM. A detailed description about how to find state invariants is given in [FL98], where four kinds of state invariants are defined: identity, state membership, uniqueness of state membership, and fixed resource. The information

[2]Since we are interested in finding macro-operators, we do not take into account self-references.

about the domain, enriched with invariants, allows to correctly unify macro-operators parameters.

To automatically build the domain hierarchy, the HIERARCHY GENERATOR requires a set of mapping functions that contain the translation rules (on types, predicates, operators, and invariants) between two adjacent levels of abstraction. These are expressed through the `:mapping` clause of the *define hierarchy* statement (see section 7.3 for further details).

Given the mapping functions, abstract operators and predicates can be generated according to a simple strategy: for each macro-operator a suitable abstract operator is generated, whose pre- and post-conditions are made coincident with those of the selected macro-operator; predicates at the abstract level are the same of the ground level, except for those not involved in any pre- or post-condition of the abstract operators.

As a final comment, let us point out that the approach described above can be used also for generating abstractions tailored to a given problem; this can be done by simply adding a dummy operator representing the goal(s) of the problem itself. The "goal-oriented" operator has only preconditions (its set of effects being empty), representing a logic conjunct of predicates that characterize the goal of the input problem. As a consequence, all sequences deemed relevant to solve the problem are easily put into evidence (as they end with the "goal-oriented" operator).

8.2.2 Notes on macro-operator abstraction

The impact of abstraction on the time spent to search for a solution of a planning problem can be positive or negative, depending on several factors -including the average branching factor, and the plan length (see [ACV04a]). Intuitively, in the worst case, the search time grows with the average branching factor (b) and the plan length (l) proportionally to b^l. Let us note that, b is influenced by the number of domain operators, the number of parameters of each operator, and the adopted heuristic function; whereas l is influenced by the problem complexity.

Typically, the abstract domain contains fewer operators than the ground domain; nevertheless, there is a usually-negative impact on the average branching factor, due to the increased complexity of the macro-operators. In fact, a macro-operator has generally a number of parameters greater than the ones belonging to each of its operators. On the other hand, using macro-operators reduces the average plan length. Thus, the time required to search for a solution at the abstract level (T_a) may be significantly lower than the time required at the ground level (T_g).

Let us recall that the abstract level is used to guide the search at the ground level. Given a plan at the abstract level, each abstract operator must be refined, each refinement becoming a planning problem at the ground level. For the sake of simplicity, let us suppose that the time required to solve a

problem using a two-level abstraction (T_h) is $T_a + T_r$, where T_r, i.e. the time needed to perform all the refinements, is proportional to $l_a \cdot b_g^{l_g/l_a}$. If T_h is greater than T_g, the impact of abstraction is negative, especially if a large number of refinements occurs. It is worth noting that when b_g is close to 1, T_r becomes greater than T_g. This typically occurs when a planner equipped with a good heuristic function is used to refine the abstract solution, thus nullifying the advantages of abstraction. On the other hand, the more b_g increases, the more abstraction becomes effective. In short, the use of abstraction based on macro-operators is not only influenced by the branching factor and the plan length, but also by the adopted planning algorithm.

To verify whether an abstraction based on macro-operator can improve the performances of the search, we made experiments on some classical benchmarking domains. Abstraction hierarchies have been automatically generated using the *DHG* system, which follows an "a priori" approach. For the sake of simplicity, only two relevant domains have been selected, i.e., *elevator* and *blocks-world*.

Let us consider the *elevator* domain, which has been described in the previous chapter. An example of recurrent sequence is *up;board*, and the corresponding macro-operator is:

```
(:action up-board
  :parameters
    (?passenger1 - passenger ?floor2 ?floor1 - floor)
  :precondition
    (and (origin ?passenger1 ?floor2)
         (lift-at ?floor1) (above ?floor1 ?floor2))
  :effect
    (and (lift-at ?floor2) (boarded ?passenger1)
         (not (lift-at ?floor1))))
```

Note that the *up;board* macro-operator has three parameters, whereas both *up* and *board* have two parameters. In this case, the number of macro-actions corresponding to the *up;board* macro-operator is greater than the number of actions corresponding to the *up* operator plus the number of actions corresponding to the *board* operator. In fact, the number of actions grows with respect to the number of objects belonging to the problem. Let n_f be the number of floors and n_p the number of passengers, the corresponding number of *up;board* instances is $n_p \cdot n_f^2$, the number of *up* instances is n_f^2, and the number of *board* instances is $n_p \cdot n_f$. Hence, the number of applicable actions depends on the number of passengers and floors belonging to the problem to be solved. The automatic hierarchy found by *DHG* has an abstract domain composed by four abstract operators (obtained from the macro-operators corresponding to the sequences *up;board*, *up;depart*, *down;board*,

and *down;depart*). Each abstract operator has three parameters (two floors and one passenger), being $4 \cdot n_f^2 \cdot n_p$ the number of applicable actions at the abstract level. On the other hand, the number of applicable actions at the ground level is $2 \cdot n_f^2 + 2 \cdot n_p n_f$. Comparing the two expressions, it is clear that the branching factor at the abstract level is greater than the one at the ground level.

As for the *blocks-world* domain, the automatic hierarchy found by *DHG* has an abstract domain composed by two abstract operators (obtained from the macro-operators corresponding to the sequences *pick-up;stack* and *unstack;put-down*). Each abstract operator has two parameters, being $2 \cdot n_b^2$ the number of applicable actions at the abstract level, where n_b is the number of blocks. On the other hand, the number of applicable actions at the ground level is $2 \cdot n_b + 2 \cdot n_b^2$. Comparing the two expressions, it can be noted that the branching factor at the abstract level is always lower than the one at the ground level.

It is now clear that different behaviors hold, depending on the characteristics of the domain taken into account. In particular, two relevant and different cases have been briefly discussed, pointing to the theoretical and actual branching factor. Roughly speaking, we expect that a hierarchical planner based on macro-operators performs better in the *blocks-world* than in the *elevator* domain.

8.3 Experiments and Results

To assess the functionality of the *DHG* system, we compared the automatically generated domain hierarchies with the corresponding domain hierarchies hand-coded by a knowledge engineer, and characterized by mapping on types, predicates, and operators. A set of benchmarking domains, taken from the planning competitions ([Lon98], [Bac00], [Lon02]), has been selected to generate the abstraction hierarchies. The domain hierarchies have been used as input for the *HW[ ]* system, which has been described in chapter 7, devised to perform planning by abstraction.

Let us briefly recall that, *HW[ ]* (which stands for *Hierarchical Wrapper*) can exploit any external PDDL-compliant planner to search for solutions at any required level of abstraction.

Experiments have been performed using *FF* [HN01] as external planner, being *HW[FF]* the resulting system. Let us point out that the planner chosen to be embedded into the system scarcely affects the relevance of the experimental results. In fact, only the relative performance between the automatic and the hand-coded versions of each domain hierarchy should be directly compared. For the description about the performance of abstraction mechanisms, see section 7.4.

Experiments have been conducted on several domains including *Depots*,

Blocks-World and *Elevator (simple-miconic)*. For each domain, a set of problems has been selected to compare the performances of *HW[FF]* using the *DHG*'s domain hierarchies with those of *HW[FF]* using the hand-coded domain hierarchies. For the sake of simplicity, since it is generally a demanding work to generate by hand abstraction hierarchies having more than two levels, the experiments have been made using two-level abstraction hierarchies.

Table 8.2 summarizes the results obtained for the selected domains. The results obtained using the planner without abstraction (*FF*, in this case) are not reported, since in this section we are not concerned on comparing the performance between a planning algorithm and its hierarchical counterpart. The columns labelled *abs* and *refs* report the time (expressed in milliseconds) needed to find the solution at the abstract level and the time needed to refine it, respectively. The column labelled *tot* reports the total time spent by *HW[FF]* to solve the problem, including disk usage, conversion to/from PDDL, etc. The column labelled *steps* is reported to compare the quality of plans (in terms of the steps required to reach the goal state) between the two counterparts.

8.3.1 Depots

The abstract level found by *DHG* for the Depots domain is composed by four abstract operators, two of them (*lift* and *drop*) are identical to those defined at the ground level, while the others are obtained from the sequences *drive;load* and *drive;unload*. The hand-coded abstraction hierarchy defines two abstract-operators (obtained from the sequences *drive;unload;drop* and *drive;lift;load*), disregards the *lifting* predicate, and substitutes *depot* and *distributor* with the supertype *place* (this one being an example of abstraction on types).

Experiments show that, for the *Depots* domain, the performances of *HW[FF]* using the hierarchy found by *DHG* are in general slight worse (the difference is about 25%) than those of *HW[FF]* fed with the hand-coded hierarchy. In my opinion, the reason lies in the fact that automatically extracted hierarchy does not include abstraction on types and/or predicates, whereas the corresponding hand-coded hierarchy introduces types and predicates mappings.

8.3.2 Elevator

The abstract level found by *DHG* for the Elevator domain is composed by four abstract operators: (obtained from the sequences *up;board*, *up;depart*, *down;board*, and *down;depart*). The corresponding hand-coded hierarchy defines two abstract operators (*load* and *unload*) and disregards two predicates (*lift-at* and *above*).

The performance measured while feeding *HW[FF]* with the hierarchy found by *DHG* is about 20% worse than the one obtained by running *HW[FF]* with the hand-coded hierarchy. Also in this case the automatic hierarchy (being pure macro-operator based) lacks of mappings on types and/or predicates.

8.3.3 Blocks-world

The abstract level found by *DHG* for the *Blocks-World* domain is composed by two abstract operators: (obtained from the sequences *pick-up;stack* and *unstack;put-down*). The corresponding hand-coded hierarchy shows an abstract domain composed by the same operators, although the predicates *handempty* and *holding* have been disregarded.

In this domain, time intervals are approximately the same, since the hierarchy obtained from *DHG* is nearly identical to the one coded by hand. In fact, both of them define the abstract domain by two operators without abstracting types. The hand-coded hierarchy disregards two predicates, `(holding ?x - block)` and `(handempty)`, but this clearly does not introduce a substantial improvement, since `holding` does not appear in the preconditions and the effects of the macro-operators, and there is no macro-operator that negates the `handempty` predicate.

8.3.4 Driver-Depots: a more complex domain

Experimental results have shown that abstraction is more effective when the complexity of planning problems increases. To assess the advantages of using this approach, a more complex domain has been devised by extending the *depot* domain taken from the AIPS 2002 planning competition [Lon02]. The *depot* domain joins two well-known planning domains: *logistics* and *blocks-world*. They have been combined to form a domain in which trucks can transport crates around, to be stacked onto pallets at their destinations. The stacking is achieved using hoists, so that the resulting stacking problem is very similar to a blocks-world problem with hands. Trucks behave like "tables", since the pallets on which crates are stacked are limited. The proposed domain extends the *depots* domain adding to it a driver able to move trucks among places (see 8.3.4). Note that the driver could simulate the behavior of an agent able to deliver objects by driving trucks from a location to another.

8.3.4 shows the hierarchy for the driverdepotsdomain, whereas 8.3.4 shows the abstract domain, obtained by applying the mapping rules to the ground domain.

Bearing in mind that the same mechanism has been applied to all operators' pairs, let us concentrate for instance on the relation that holds between

```
(define (domain driverdepots-ground)
  (:requirements :strips :typing)
  (:types
    place locatable - object depot distributor - place
    pallet crate - surface
    driver truck hoist surface - locatable)
  (:predicates
    (at ?l - locatable ?p - place) (on ?c - crate ?s - surface)
    (in ?c - crate ?t - truck) (lifting ?h - hoist ?c - crate)
    (available ?h - hoist) (clear ?s - surface)
    (driving ?d - driver ?t - truck) (empty ?t - truck))
  (:action Drive
    :parameters (?t - truck ?p1 ?p2  place ?d - driver)
    :precondition (and (at ?t ?p1) (driving ?d ?t))
    :effect (and (not (at ?t ?p1)) (at ?t ?p2)))
  (:action Lift
     :parameters (?h - hoist ?p  place ?c - crate ?s - surface)
     :precondition (and (at ?h ?p) (available ?h) (at ?c ?p)
                        (on ?c ?s) (clear ?c))
     :effect (and (not (at ?c ?p)) (clear ?s) (lifting ?h ?c)
                  (not (clear ?c)) (not (available ?h))
                  (not (on ?c ?s))))
  (:action Drop
    :parameters (?h - hoist ?c  crate ?s - surface ?p - place)
    :precondition (and (at ?h ?p) (at ?s ?p) (clear ?s)
                       (lifting ?h ?c))
    :effect (and (available ?h) (at ?c ?p)
                 (not (lifting ?h ?c)) (not (clear ?s))(clear ?c)
                 (on ?c ?s)))
  (:action Load
    :parameters (?h - hoist ?c - crate ?t  truck p - place)
    :precondition (and (at ?h ?p) (at ?t ?p) (lifting ?h ?c))
    :effect (and (not (lifting ?h ?c)) (in ?c ?t) (available ?h)))
  (:action Unload
    :parameters (?h - hoist ?c - crate ?t - truck ?p - place)
    :precondition (and (at ?h ?p) (at ?t ?p) (available ?h)
                       (in ?c ?t))
    :effect (and (not (in ?c ?t)) (not (available ?h))
                 (lifting ?h ?c)))
  (:action Board
    :parameters (?d - driver ?t - truck ?p - place)
    :precondition (and (at ?t ?p) (at ?d ?p) (empty ?t))
    :effect (and (not (at ?d ?p)) (driving ?d ?t)
                 (not (empty ?t))))
  (:action Disembark
    :parameters (?d - driver ?t - truck ?p - place)
    :precondition (and (at ?t ?p) (driving ?d ?t))
    :effect (and (not (driving ?d ?t)) (at ?d ?p) (empty ?t)))
  (:action Walk
    :parameters (?d - driver ?p1 ?p2 - place)
    :precondition (and (at ?d ?p1))
    :effect (and (not (at ?d ?p1)) (at ?d ?p2))))
```

```
(define (domain driverdepots-abstract)
  (:requirements :strips :typing)
  (:types place locatable - object
          driver truck surface - locatable
          pallet crate - surface)
  (:predicates (at ?l - locatable ?p - place)
               (on ?c - crate ?s - surface)
               (in ?c - crate ?t - truck)
               (clear ?s - surface)
               (driving ?d - driver ?t - truck)
               (empty ?t - truck))

  (:action DriveUnloadDrop
    :parameters (?t - truck ?p1 ?p2 - place ?d - driver
                 ?c - crate ?s - surface)
    :precondition (and (at ?t ?p1) (driving ?d ?t) (in ?c ?t)
                       (at ?s ?p2) (clear ?s))
    :effect
      (and (not (at ?t ?p1)) (at ?t ?p2) (not (in ?c ?t))
           (at ?c ?p2) (not (clear ?s)) (clear ?c) (on ?c ?s)))
  (:action LiftLoad
    :parameters (?c - crate ?t - truck ?s - surface ?p - place)
    :precondition (and (at ?c ?p) (on ?c ?s) (clear ?c)
                       (at ?t ?p))
    :effect
      (and (not (at ?c ?p)) (not (clear ?c)) (clear ?s)
           (in ?c ?t) (not (on ?c ?s))))
  (:action WalkBoard
    :parameters (?d - driver ?t - truck ?p1 ?p2 - place)
    :precondition (and (at ?t ?p2) (at ?d ?p1) (empty ?t))
    :effect (and (not (at ?d ?p1)) (driving ?d ?t) (not (empty ?t))))
  (:action DriveDisembark
    :parameters (?d - driver ?t - truck ?p1 ?p2 - place)
    :precondition (and (at ?t ?p1) (driving ?d ?t))
    :effect
      (and (not (driving ?d ?t)) (at ?d ?p2) (empty ?t)
           (not (at ?t p1)) (at ?t ?p2))
  (:action Drive
    :parameters (?t - truck ?p1 ?p2 - place ?d - driver)
    :precondition (and (at ?t ?p1) (driving ?d ?t))
    :effect (and (not (at ?t ?p1)) (at ?t ?p2)))))
```

Figure 8.5: The *driverdepots-abstract* domain.

```
(define (hierarchy driverdepots)
  (:domains
    driverdepots-ground
    driverdepots-abstract)
  (:mapping
    (driverdepot-ground
     driverdepot-abstract)
    :types
      ((place depot)
       (place distributor)
       (nil hoist))
    :predicates
      ((nil
         (lifting ?h - hoist ?c - crate))
       (nil
         (available ?h - hoist))
       (nil
         (at ?h - hoist ?p - place)))
    :actions
      ((nil (load ?h ?c ?t ?p))
       (nil (unload ?h ?c ?t ?p)
       (nil (lift ?h ?c ?s ?p))
       (nil (drop ?h ?c ?s ?p))
       (nil (walk ?d ?p1 ?p2))
       (nil (board ?d ?t ?p))
       (nil (disembark ?d ?t ?p))
       (drive-unload-drop
           ?t ?p1 ?p2 ?d ?c ?s)
          (and (drive ?t ?p1 ?p2 ?d)
             (unload ?h ?c ?t ?p2)
             (drop ?h ?c ?s ?p2)))
       ((walk-board ?d ?p1 ?p2 ?t)
         (and (walk ?d ?p1 ?p2)
             (board ?d ?t ?p2)))
       ((drive-disembark ?d ?t ?p1 ?p2)
         (and (drive ?t ?p1 ?p2 ?d)
             (disembark ?d ?t ?p2)))
       ((lift-load ?c ?t ?s ?p)
         (and (lift ?h ?p ?c ?s)
             (load ?h ?c ?t ?p)))))
```

Figure 8.6: Hierarchy definition for the *driverdepots* domain.

`drive` (source node) and `board` (destination node). Considering that the effects of the drive operator are:

```
(not (at ?t ?p1)) (at ?t ?p2)
```

and that the preconditions of the board operator are:

```
(at ?t ?p) (at ?d ?p) (empty ?t)
```

we label the corresponding edge with the pair $< 11 >$. In fact, it is apparent that `drive` establishes one precondition for `board`, while negating another. shows the resulting graph for the *driverdepot* domain after the pruning activity. The resulting macro-operator schemata are (; being used for concatenation): *drive;unload;drop*, *drive;load;lift*, *drive;disembark*, *lift;load*, *drop;unload*, *load;lift*, *unload;drop*, and *walk;board*. Among these, *load;lift*, *drive;load;lift*, and *drop;unload* have been disregarded since they become meaningless when applied to the same object. For instance, loading a truck with a crate C and then lifting C back does not alter the state of the world. Hence, *drive;unload;drop*, *unload;drop*, *walk;board*, *drive;disembark*, and *lift;load* are the selected macro-operator schemata. As for the generation of abstract-operators, let us note that *drive;unload;drop* and *unload;drop*, can be considered alternative refinements of the same abstract-operator. Furthermore, let us stress that the `lifting` predicate does not appear as precondition or effect in any abstract operator; hence, it can be removed at the abstract level. Since we are interested in abstracting the domain on types, predicates, and operators, the type hierarchy could be simplified by deleting for example- the `hoist` type. This choice is feasible because hoists are always available, in every place (consequently, the available predicate can also be removed). Moreover, the type hierarchy can be further reduced by considering both `distributors` and `depots` as generic `places`.

8.4 Summary

The automatic definition of macro-operators is one the most important steps in the task of abstracting a planning domain. In this chapter, a technique devised to tackle this problem has been described, its implementation yielding a system called *DHG* (standing for *Domain-oriented Hierarchy Generator*).

The process consists of finding a set of relevant sequences and then promoting them to macro-operators using a graph-oriented technique. A directed graph, containing information about the dependencies between domain operators, is built. Nodes represent operators, and edges represent relations between effects of the source node and preconditions of the destination node. A relevant sequence of operators may be extracted from each acyclic path. As considering all possible paths would end up to a large amount of sequences, G is pruned through a set of domain-independent

heuristics. A set of macro-operators is then generated from a selected set of sequences. To avoid semantic inconsistencies, an analysis aimed at finding state invariants is also performed.

Experimental results obtained comparing the performances of the hand-coded and automatically-generated abstraction hierarchies are encouraging and demonstrate the validity of the approach. In particular, the system is able to identify suitable macro-operators, used as starting point for populating the abstract level. Such macro-operators usually represent good alternatives to those extracted by a knowledge engineer after a thorough (and sometimes painful!) domain analysis. The slightly negative impact on performances obtained by resorting to the automatic generation of abstraction hierarchies is more than counterbalanced by the fact that a negligible effort is required to the knowledge engineer in order to obtain suitable abstractions. The environment used to perform the experiments combines *DHG* with *HW[FF]*. The latter is a (parametric) hierarchical planning environment able to embed and run an external planner –in this case *FF*– at different levels of granularity.

We are currently experimenting a further approach to improve the performance of planners, which exploits control knowledge to guide the search. In particular, the system called HW[IPSS] has been implemented (see [ACVF05]). It combines the advantages of abstraction with the use of explicit control rules in planning. In this context, the adoption of abstraction techniques remarkably simplify the process of acquiring knowledge done by the HAMLET system, an incremental learning system based on Explanation Based Learning (EBL) and inductive refinement of control rules (described in [BV97]).

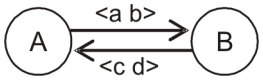
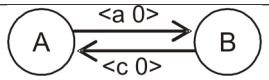
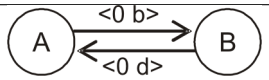
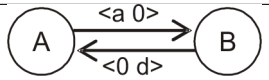
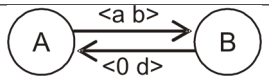
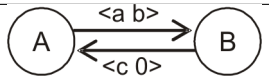
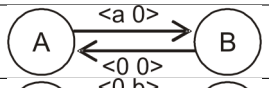
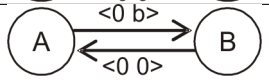
Type	Relationships	Supporting Evidence	Action
1		(i) if $a = c$ and $b = d$ there is no supporting evidence for assuming that A usually precedes B in a plan, and vice-versa;	remove both edges.
		(ii) if $a > c$ there is a high likelihood that A precedes B;	remove top edge.
		(iii) if $c > a$, there is a high likelihood that B precedes A.	remove bottom edge.
2		(i) if $a > c$ there is a high likelihood that A precedes B;	remove top edge.
		(ii) if $c > a$, there is a high likelihood that B precedes A	remove bottom edge.
3		$A(B)$ negates one or more preconditions required by $B(A)$.	remove both edges.
4		B negates one or more preconditions required by A.	remove bottom edge.
5		B negates one or more preconditions required by A.	remove both edges.
6		A and B are usually complementary or loosely-coupled actions.	remove both edges.
7		A precedes B with high likelihood	remove top edge.
8		A negates one or more preconditions required by B	remove top edge.

Table 8.1: Heuristics for pruning the operators' graph.

Table 8.2: Hand-coded vs automatically generated hierarchy performance comparison using *HW[FF]*.

Problem	**Hand-Coded**				**Automatic**			
	abs	*ref*	*tot*	*steps*	*abs*	*ref*	*tot*	*steps*
Depot1	28	73	106	12	23	120	147	11
Depot2	54	128	187	17	33	207	245	17
Depot3	488	340	841	38	69	532	609	36
Depot4	292	416	717	43	389	581	982	31
Depot5	845	100	950	71	-	-	-	-
Elevator1	10	51	63	8	19	57	78	8
Elevator2	17	142	163	15	20	145	170	16
Elevator3	18	226	248	4	11	28	40	4
Elevator4	18	359	383	23	23	396	427	26
Elevator5	19	740	767	28	26	566	603	28
Blocks1	11	41	54	6	11	41	54	6
Blocks2	18	104	125	14	19	107	129	14
Blocks3	40	450	497	44	41	463	513	44
Blocks4	45	479	532	48	46	471	524	48
Blocks5	55	501	564	48	58	472	538	48

Macro-Operator Schema	**Ground Sequence**	**Preconditions**	**Effects**
(DriveUnloadDrop ?h - hoist ?t - truck ?p1 ?p2 - place ?d - driver ?c - crate ?s - surface)	drive; unload; drop	(at ?t ?p1) (driving ?d ?t) (in ?c ?t) (at ?s ?p2) (clear ?s) (at ?h ?p2) (available ?h)	(not (at ?t ?p1)) (at ?t ?p2) (at ?c ?p) (at ?c ?p2) (not (clear ?s)) (clear ?c)
(UnloadDrop ?h - hoist ?t - truck ?c - crate ?s - surface)	unload; drop	(at ?t ?p) (in ?c ?t) (clear ?s) (at ?h ?p) (available ?h)	(not (in ?c ?t)) (at ?c ?p) (clear ?c) (on ?c ?s)
(WalkBoard ?d - driver ?p1 ?p2 - place ?t - truck)	walk; board	(at ?d ?p1) (at ?t ?p2) (empty ?t)	(not (at ?d ?p1)) (driving ?d ?t) (not (empty ?t))
(DriveDisembark ?d - driver ?t - truck ?p1 ?p2 - place)	drive; disembark	(at ?t ?p1) (driving ?d ?t) (empty ?t)	(not (driving ?d ?t)) (at ?d ?p2) (at ?t ?p2) (not (at ?t ?p1))
(LiftLoad ?h - hoist ?c - crate ?t - truck ?s - surface ?p - place)	lift; load	(at ?h ?p) (available ?h) (at ?c ?p) (on ?c ?s) (clear ?c) (at ?t ?p)	(not (at ?c ?p)) (not (clear ?c)) (clear ?s) (in ?c ?t) (not (on ?c ?s))

Table 8.3: Selected macro-operator schemata for the *Driver-depot* domain.

Conclusions

Chapter 9

Conclusions and Future Work

In this thesis, abstraction techniques for managing complexity in agent systems and planning have been investigated.

The research presented in this thesis was motivated by the need to cope with applications for modern computing and information processing systems. These applications have in common that they are inherently distributed in data and information to be processed, and that are inherently complex, since they are too large to be solved by a single, centralized system because of limitations available at a given level of hardware or software technology. Multiagent systems offer a promising and innovative way to understand, manage, and use distributed, large-scale, dynamic, open, and heterogeneous computing and information systems. The Internet is the most prominent example of such systems.

The contributions described in this thesis are the generic multi-agent architecture *PACMAS*, the parametric hierarchical wrapper *HW[ ]*, and the domain hierarchy generator *DHG*.

The PACMAS architecture has been designed to support the implementation of applications aimed at: (i) retrieving heterogeneous data spread among different sources, (ii) filtering and organizing them according to personal interests explicitly stated by each user, and (iii) providing adaptation techniques to improve and refine throughout time the profile of each selected user.

HW[ ] has been devised and implemented to perform planning by abstraction that is an effective approach for implementing the planning capabilities of an intelligent agent.

DHG has been devised to automatically generate abstraction hierarchies to be used by HW[ ] to plan hierarchically.

In part I, the basic issues surrounding the design and implementation of intelligent agents, together with the aspects regarding their pro-active

capability, and abstraction techniques that can be exploited to improve their planning performances, have been illustrated.

In part II, the PACMAS architecture has been presented in detail. To highlight the peculiarities and the effectiveness of the proposed architecture, two relevant case studies implemented exploiting the PACMAS architecture, have been described: the first one is focused on giving a support to undergraduate and graduate students in their university activities; the second one is devoted to create press-reviews from online newspapers through the classification of newspaper articles. Successfull tests on the developed applications demonstrated the effectiveness of the architecure.

In part III, the parametric system *HW[ ]* for planning by abstraction, and the *DHG* system for automatically generate abstraction hierarchies, have been presented, together with experiments on a set of benchmarking domains. Experimental results highlight that abstraction is useful for improving the performances of classical planners. Moreover, a direct comparison between the performances of automatically-generated versus hand-coded abstraction hierarchies demonstrates the validity of the approach.

As for the future work, several improvements of the PACMAS architecture are under investigation. In particular, new algorithms to enhance the adaptive capabilities of the involved agents, are in development. Furthermore, the integration of the planning system HW[ ] within PACMAS agents, to give them powerful planning capabilities, is under study.

Bibliography

[ACMV05] Giuliano Armano, Giancarlo Cherchi, Andrea Manconi, and Eloisa Vargiu. PACMAS: A Personalized, Adaptive, and Cooperative MultiAgent System Architecture. In *Proceedings of Workshop dagli Oggetti agli Agenti, Simulazione e Analisi Formale di Sistemi Complessi (WOA 2005)*, Camerino (MC) Italy, November 2005.

[ACV03a] Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. A Parametric Hierarchical Planner for Experimenting Abstraction Techniques. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI '03)*, pages 936–941, 2003.

[ACV03b] Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. An Extension to PDDL for Hierarchical Planning. In *Proceedings of Workshop on PDDL (ICAPS'03)*, pages 1–6, Trento (Italy), 2003.

[ACV03c] Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. Experimenting the Performance of Abstractions Mechanisms through a Parametric Hierarchical Planner. In *Proceedings of IASTED International Conference on Artificial Intelligence and Applications(AIA 2003)*, pages 399–404, Innsbruck (Austria), 2003.

[ACV03d] Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. Generating Abstractions from Static Domain Analysis. In *Proceedings of Workshop dagli Oggetti agli Agenti - Sistemi Intelligenti e Computazione Pervasiva (WOA'03)*, pages 140–147, Cagliari (Italy), 2003.

[ACV03e] Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. HW[ ]: A Parametric System for Planning by Abstraction. *PLANET News*, 6:5–10, 2003.

[ACV03f] Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. Planning by Abstraction Using HW[ ]. *AI*IA 2003: Advances in Artificial Intelligence*, LNAI 2829:349–361, 2003.

[ACV04a] Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. A Critical Look at the Abstraction Based on Macro-Operators. In *Proceedings of the 3rd Italian Workshop on Planning and Scheduling (AIxIA 2004)*, Perugia (Italy), September 2004.

[ACV04b] Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. Automatic Generation of Macro-Operators from Static Domain Analysis. In *Proceedings of ECAI 2004*, pages 955–956, Valencia (Spain), 2004.

[ACV05] Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. DHG: A System for Generating Macro-Operators from Static Domain Analysis. In *Proceedings of the International Conference on Artificial Intelligence and Applications (AIA05)*, 2005.

[ACVF05] Giuliano Armano, Giancarlo Cherchi, Eloisa Vargiu, and Susana Fernandez. Integrating Abstraction Techniques and EBL Control Rules to Perform Automated Planning. In *Proceedings of PLANSIG 2005, the 24th Annual Workshop of the UK Planning and Scheduling Special Interest Group*, City University, London, UK, December 2005.

[ADW94] Chidanand Apte, Fred Damerau, and Sholom M. Weiss. Automated learning of decision rules for text categorization. *Information Systems*, 12(3):233–251, 1994.

[AFJM95] Robert Armstrong, Dayne Freitag, Thorsten Joachims, and Tom Mitchell. Webwatcher: A learning apprentice for the world wide web. In *AAAI Spring Symposium on Information Gathering*, pages 6–12, 1995.

[Ama68] Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence*, volume 3, pages 131–171. Elsevier/North-Holland, Amsterdam, London, New York, 1968.

[AV01] Giuliano Armano and Eloisa Vargiu. An adaptive approach for planning in dynamic environments. In *Proceedings of International Conference on Artificial Intelligence (IC-AI 2001)*, pages 987–993, Las Vegas (Nevada), 2001.

[Bac00] F. Bacchus. Results of the aips 2000 planning competition, 2000. Url: http://www.cs.toronto.edu/aips2000.

[BF95] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[BG99] Blai Bonet and Hector Geffner. Planning as heuristic search: New results. In *ECP*, pages 360–372, 1999.

[BIP91] Michael E. Bratman, David Israel, and Martha Pollack. Plans and resource-bounded practical reasoning. In Robert Cummins and John L. Pollock, editors, *Philosophy and AI: Essays at the Interface*, pages 1–22. The MIT Press, Cambridge, Massachusetts, 1991.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, California, 2 edition, 1994.

[BPR00] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing multi-agent systems with jade. In *Eventh International Workshop on Agent Theories, Architectures, and Languages (ATAL-2000)*, 2000.

[Bra87] Michael Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.

[Bro85] Rodney A. Brooks. A robust layered control system for a mobile robot. Technical report, Cambridge, MA, USA, 1985.

[BS04] Adi Botea, Martin Müller 0003, and Jonathan Schaeffer. Using component abstraction for automatic generation of macro-actions. In *ICAPS*, pages 181–190, 2004.

[BV97] D. Borrajo and M. M. Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning*, 11(1-5):371–405, February 1997.

[BW95] Ralph Bergmann and Wolfgang Wilke. Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research (JAIR)*, 3:53–118, 1995.

[CCGJ04] Ricardo Carreira, Jaime M. Crato, Daniel Gonalves, and Joaquim A. Jorge. Evaluating adaptive user profiles for news classification. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*, pages 206–212, New York, NY, USA, 2004. ACM Press.

[CDMV05] Giancarlo Cherchi, Dario Deledda, Andrea Manconi, and Eloisa Vargiu. Text categorization using a personalized, adaptive, and cooperative multiagent system. In *Proceedings of Workshop*

dagli Oggetti agli Agenti, Simulazione e Analisi Formale di Sistemi Complessi (WOA 2005), Camerino (MC) Italy, November 2005.

[CS93] Scott Cost and Steven Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–78, 1993.

[CS96] William W. Cohen and Yoram Singer. Context-sensitive learning methods for text categorization. In Hans-Peter Frei, Donna Harman, Peter Schaauble, and Ross Wilkinson, editors, *Proceedings of SIGIR-96, 19th ACM International Conference on Research and Development in Information Retrieval*, pages 307–315. ACM Press, New York, US, 1996.

[DSW97] K. Decker, K. Sycara, and M. Williamson. Middle-agents for the internet. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 97)*, pages 578–583, 1997.

[EH03] Stefan Edelkamp and Jorg Hoffmann. Pddl2.2: The language for the classical part of the 4th international planning competition. Technical report, Institut fur Informatik, Universitt Freiburg, Germany, 2003.

[EHN94] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1123–1128, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[EW95a] Andreas S. Weigend Erik Wiener, Jan O. Pedersen. A neural network approach to topic spotting. In *Proceedings of 4th Annual Symposium on Document Analysis and Information Retrieval*, pages 317–332, Las Vegas, US, 1995.

[EW95b] O. Etzioni and D. Weld. Intelligent agents on the internet: fact, fiction and forecast. *IEEE Expert*, 10(4):44–49, 1995.

[Fer92] I. A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents.* PhD thesis, Cambridge, UK, 1992.

[Fis94] Michael Fisher. A survey of concurrent METATEM – the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic - Proceedings of the First Intemational Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Heidelberg, Germany, 1994.

[FL98] M. Fox and D. Long. The automatic inference of state invariants in tim. *Journal of Artificial Intelligence Research (JAIR)*, 9:367–421, 1998.

[FL03] M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[FN71] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[GAV05] G. Cherchi M. Colombetti A. Gerevini M. Mari A. Poggi C. Santoro E. Tramontana G. Armano, P. Baroni and M. Verdicchio. ANEMONE - A Network of Multi-Agent Platforms for Academic Communities. In *Proceedings of Workshop dagli Oggetti agli Agenti, Simulazione e Analisi Formale di Sistemi Complessi (WOA 2005)*, Camerino (MC) Italy, November 2005.

[GBH88] L. Gasser, C. Braganza, and N. Herman. Implementing distributed ai systems using mace. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 445–450. Kaufmann, San Mateo, CA, 1988.

[GL05] Alfonso Gerevini and Derek Long. Plan constraints and preferences in pddl3. Technical report, Department of Electronics for Automation, University of Brescia, Italy, 2005.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[GS96] A. Gerevini and L. Schubert. Accelerating Partial-Order Planners: Some Techniques for Effective Search Control and Pruning. *Journal of Artificial Intelligence Research (JAIR)*, 5:95–137, 1996.

[GS02] Alfonso Gerevini and Ivan Serina. Lpg: A planner based on local search for planning graphs with action costs. In *AIPS*, pages 13–22, 2002.

[GSF+04] J.A. Giampapa, K. Sycara, A. Fath, A. Steinfeld, and D. Siewiorek. A multi-agent system for automatically resolving network interoperability problems. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1462–1463, 2004.

[GW90] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. Technical Report 9001-14, IRST, Trento, Italy, 1990.

[HN00] J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302, 2000.

[HN01] J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302, 2001.

[HRWL83] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat. An overview of expert systems. In F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, editors, *Building Expert Systems*, pages 3–29. Addison-Wesley, London, 1983.

[HSM94] H. Haugeneder, D. Steiner, and F. G. McCabe. IMAGINE: A framework for building multi-agent systems. In S. M. Deen, editor, *Proceedings of the 1994 International Working Conference on Cooperating Knowledge Based Systems (CKBS-94)*, pages 31–64, UK, 1994.

[Jac86] Peter Jackson. *Introduction to expert systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[JhMC+96] Nick R. Jennings, E. h. Mamdani, Jose Manuel Corera, Inaki Laresgoiti, Fabien Perriollat, Paul Skarek, and Laszlo Zsolt Varga. Using archon to develop real-world dai applications, part 1. *IEEE Expert: Intelligent Systems and Their Applications*, 11(6):64–70, 1996.

[KAH94] Craig A. Knoblock, Yigal Arens, and Chun-Nan Hsu. Cooperating agents for information retrieval. In *Proceedings of the Second International Conference on Cooperative Information Systems*, Toronto, Ontario, Canada, 1994. University of Toronto Press.

[Kno91] C. Knoblock. Automatically generating abstractions for problem solving, 1991.

[Kno94] C.A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[Kor87] R.E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[KP98] R. Kohavi and F. Provost. Glossary of terms. *Special issue on applications of machine learning and the knowledge discovery process, Machine Learning*, 30(2/3):271–274, 1998.

[Kra97] J. Kramer. Agent based personalized information retrieval, 1997.

[KS98] H. Kautz and B. Selman. *The Role of Domain Specific Knowledge in the Planning as Satisfiability Framework*, pages 181–189. Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98). 1998.

[Lie95] Henry Lieberman. Letizia: An agent that assists web browsing. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 924–929, Montreal, Quebec, Canada, 1995. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[Lie97] H. Lieberman. Autonomous interface agents. In *Proceedings of the ACM Conference on Computers and Human Interface (CHI-97)*, pages 67–74, 1997.

[LKRH90] E. Lutz, H.V. Kleist-Retzow, and K. Hoernig. Mafiaan active mail-filter-agent for an intelligent document processing support. *ACM SIGOIS Bulletin*, 11(4):16–32, 1990.

[Lon98] D. Long. The aips-98 planning competition. *AI Magazine*, 21(2):13–33, 1998.

[Lon02] D. Long. Results of the aips 2002 planning competition, 2002. Url: http://www.dur.ac.uk/d.p.long/competition.html.

[LR94] David D. Lewis and Marc Ringuette. A comparison of two learning algorithms for text categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 81–93, Las Vegas, US, 1994.

[Mae94] P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):31–40, 1994.

[MCd96] Bernard Moulin and Brahim Chaib-draa. An overview of distributed artificial intelligence. pages 3–55, 1996.

[McD98] Drew McDermott. Pddl — the planning domain definition language, 1998.

[McD05] Drew McDermott. Opt manual version 1.7.3, draft, 2005.

[MRG96] I. Moulinier, G. Raskinis, and J.-G. Ganascia. Text categorization: a symbolic approach. In *Proceedings of 5th Annual Symposium on Document Analysis and Information Retrieval*, pages 87–99, Las Vegas, US, 1996.

[Mül96] Jörg P. Müller. A cooperation model for autonomous agents. In *ATAL*, pages 245–260, 1996.

[NS72] A. Newell and H.A. Simon. *Human Problem Solving*. Prentice-HAll, Englewood Cliffs, NJ, 1972.

[Nwa96] H. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, 1996.

[Ped89] E. P. D. Pednault. Adl: Exploring the middle ground between strips and the situation calculus. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *KR'89: Proc. of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332. Kaufmann, San Mateo, CA, 1989.

[Por80] M.F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[PW92] J. S. Penberthy and D. S. Weld. Ucpop: A sound, complete, partial order planner for adl. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, pages 103–114. Kaufmann, San Mateo, CA, 1992.

[Rao96] Anand S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *MAAMAW*, pages 42–55, 1996.

[Rei80] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Sac74] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[SM03] B. Sheth and P. Maes. Evolving agents for personalized information filtering. In IEEE Press, editor, *9th Conference on Artificial Intelligence for Applications (CAIA-93)*, pages 345–352, 2003.

[Tat74] A. Tate. Interplan: A plan generation system which can deal with interactions between goals. Research Memorandum MIP-R-109, Machine Intelligence Research Unit, University of Edinburgh, 1974.

[Ten88] Josh Tenenberg. *Abstraction in Planning*. PhD thesis, Computer Science Department, University of Rochester, 1988.

[TH93] Konstadinos Tzeras and Stephan Hartmann. Automatic indexing based on Bayesian inference networks. In Robert Korfhage, Edie Rasmussen, and Peter Willett, editors, *Proceedings of SIGIR-93, 16th ACM International Conference on Research and Development in Information Retrieval*, pages 22–34, Pittsburgh, US, 1993. ACM Press, New York, US.

[Tho93] Sarah Rebecca Thomas. *PLACA, an agent oriented programming language*. PhD thesis, Stanford, CA, USA, 1993.

[WAL77] R. WALDINGER. *Achieving several goals simultaneously*, pages 94–136. Ellis Horwood, Chichester, England, 1977.

[War76] D. Warren. Warplan: A system for generating plans. Memo 76, Department of Computational Logic, University of Edinburgh, June 1976.

[Whi] James E. White. Telescript technology: Mobile agents. Also available as General Magic White Paper.

[WJ95] M. Wooldridge and N.R. Jennings. *Intelligent Agents*, chapter Agent Theories, Architectures, and Languages: a Survey, pages 1–22. Berlin: Springer-Verlag, 1995.

[Woo02] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley and Sons, Chichester, England, 2002.

[Yan99] Y. Yang. An evaluation of statistical approaches to text categorization. *Information Retrieval*, 1(1/2):69–90, 1999.

[YC94] Y. Yang and C.G. Chute. An example-based mapping method for text categorization and retrieval. *ACM Transactions on Information Systems*, 12(3):252–277, 1994.

[YL99] Yiming Yang and Xin Liu. A re-examination of text categorization methods. In Marti A. Hearst, Fredric Gey, and Richard Tong, editors, *Proceedings of SIGIR-99, 22nd ACM International Conference on Research and Development in Information Retrieval*, pages 42–49, Berkeley, US, 1999. ACM Press, New York, US.

[YP97] Y. Yang and J. O. Pedersen. A comparative study on feature selection in text categorization. In *International Conference on Machine Learning*, pages 412–420, 1997.