Planning
and
the Belief-Desire-Intention Model of Agency

by

Andrzej Walczak

Thesis

Presented to the Faculty of Informatics of the

University of Hamburg

in Partial Fulfillment

of the Requirements

for the Degree of

Diplom-Infomatiker

Supervisor: Prof. Dr. W. Lamersdorf

Cosupervisor: Dr. D. Moldt

University of Hamburg

April 2005

Acknowledgments

I wish to thank Alexander Pokahr and Lars Braubach for supporting my aims during this project. Thank you, for providing me with the agent platform and BDI engine where the concepts of this thesis could be developed and tested with, and for introducing me to the great agent programming paradigm. A great job in making this thesis readable, by correcting orthographic and grammatical errors, was done by Julia Schwab and Marek Walczak.

Andrzej Walczak

University of Hamburg
April 2005

Planning
and
the Belief-Desire-Intention Model of Agency

Andrzej Walczak

University of Hamburg, 2005

Artificial intelligence researches the field of planning as a process able to provide agents with a course of action guiding them towards their aims. Agents incorporating the belief-desire-intention model of agency are generally equipped with a library of plans, avoiding the burden of planning at runtime. Nevertheless, there is a need for planning techniques within BDI agents, in order to cope with problems, for which it is difficult to create generic plans in advance.

It needs to be investigated in theory and practice, what techniques are appropriate for this purpose. How to integrate planning with the BDI model of agency and how well does planning perform in multi-agent environments. It is also interesting if a successful planner could be implemented with means of JAVA™. The planning community has developed many concepts with respect to this topic. Simple state space approaches, hierarchical task networks, partial order and deductive planning techniques can be used for the purpose of this diploma thesis. To integrate a planner with a BDI system an approach may be pursued where the planner controls the underlying BDI system or it uses BDI goals and plans to compose them into new plans in a dynamic way. On the other hand, the BDI engine could also trigger the planner, to create plans composed of specially crafted operators.

The aim of this diploma thesis project is to implement a planner in JAVA™ language that uses JAVA™ as its representation for planning concepts and to adapt and integrate this planner into the JADEX BDI reasoning engine. The applicability of modern planning techniques to multi-agent systems should also be proved. The planning techniques are investigated on planners used in the practice. They are compared to each other on the basis of efficiency, methods used and features supported. Most successful concepts are taken to design and implement a planner that would fit into a BDI agent architecture, in particular into the JADEX BDI agent system. The overall approach is evaluated on small examples.

Contents

List of Figures

Statement

I would like to assert hereby that the thesis work has been done by myself to the full length and content. All means and sources used therefore are explicitly mentioned here.

Andrzej Walczak

University of Hamburg
April 2005

Chapter 1

Introduction

This chapter introduces the subject of planning and agency. Related question will be posed in respect to the combination of artificial intelligence (AI) planning and the belief-desire-intention (BDI) model of agency. At last, the intention of the diploma thesis project will be stated and the approach taken sketched.

1.1 Topics

Both fields of planning and agency in computer science and AI have long research tradition. The notion of planning is central to the agent concept, especially from the viewpoint of AI. First agent architectures have been based on the techniques of deductive problem solving and reasoning. Planning was one of the main problem solving techniques investigated at that time.

Planning can be viewed from different perspectives, depicted by the following definitions. There are psychologically motivated definitions, which emphasize the mental process of planning, and there are definitions concerning the representation of plans being the determining factor for embedding process.

Wilensky (1983, p. 1) gives a relative wide definition of planning and processes occurring during plan execution.

> "Planning concerns the process by which people select a course of action – deciding what they want, formulating and revise plans, dealing with problems and adversity, making choices, and eventually performing some action."

Hayes-Roth & Hayes-Roth (1979, pp. 275). put the planning as a part of two stages process they call *planning and control*.

> "We define planning as the predetermination of a course of action aimed at achieving some goal. It is the first stage of a two-stage problem-solving process. The second stage entails monitoring and guiding the execution of a plan to a useful conclusion."

The perspective on planning taken in this paper borrows from the definition given in the area of AI. The classical view on planning is to understand it as a process of building a sequence of actions that will achieve a stated goal (Russell & Norvig 2003). The sequence transforms a given initial state into some state that meets the goal description. The notion of a *state* is given by the following definition:

> "The classical definition of the planning problem assumes a state-based representation of the world. This means that the world is represented by taking a 'snapshot' of it at one particular time and describing the world as it appears in this snapshot [...]" (Wilkins 1988, p. 4).

The classical view is settled in domains that are: static - as changes are implied by the plan actions only, deterministic, observable - making the process of planning fully informed, discrete (in time, actions, objects and effects) and finite (Russell & Norvig 2003). An expanded view is given in the following chapter.

Agents act. An Agent is to be understood here as a computing item. It is a paradigm melting the view of computer science - providing approved techniques for development of strong reliable software and hardware - with the view of AI - promising advanced features of being rational, self-sufficient, autonomous and flexible. Wooldridge (1997) characterizes an *agent* to be:

- an autonomous system, making decisions based on its internal state,
- situated in an environment and being able to perceive it in order to react to the changes,
- able to take the initiative and exhibit goal–directed behavior,
- able to interact with other agents and to cooperate.

An agent, being an autonomous system, has control over its internal state and its actions. It can be distinguished and is fully separable from its embedding environment, having clearly defined interfaces. It is designed to fulfill specific goals and it pursues its goals by exhibiting reactive behavior as a timely response to changes in the environment and proactive behavior as a response to internal processing, possibly in anticipation of future environmental states (Jennings 1999).

An agent acts rational if it chooses – based on its knowledge and resources available – actions promising the best expected outcome. What is the best outcome is defined by a function called *performance measure*, evaluating agent behavior in an environment. This leads to the following definition of a *rational agent*:

> "For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever build-in knowledge the agent has" (Russell & Norvig 2003).

Russell & Norvig (2003) give the definition for four types of agents. The *simple reflex agent* determines its actions solely through reactions based on *condition-action* rules. If the word percept fulfills a condition rule, it becomes active. The

agent chooses then an active rule and performs the associated action. This type of an agent has the desirable property of being simple, but it scales poorly to complex problems requiring more than one action and envisioning forthcoming states of the environment. Learning in this model of agency is done easily by creating new associations at runtime.

The *model-based agent* maintains a state of the world updated by its sensory inputs and by functions describing changes over time including the effects of own actions. This approach softens the requirements of fully observable environment posed by the former reactive agent. The structure underlying the reasoning is the representation of the environment encapsulated in agent's internal state.

Goal-based agents comprise *goals*, i.e. descriptions of desirable states used to guide their behavior. As their goals can change or be pursued by different means, goal-based agents are more flexible in the application. The choice of an action is determined by the environment, agent's internal state and by the set of goals the agent is currently pursuing. Planning and search is used if there are goals that cannot be achieved by a single action or procedure. The JADEX – architecture allows for creating goal-based agents.

Many successful agent architectures are based on the belief-desire-model of agency that may be seen as an extension to the *goal-based* agent type described above. It has been devised by Bratman (1987) as philosophical means to explain human intentional behavior and will be described further in Chapter 4.

Fourth agent type determines its actions on the basis of a *utility function* that sums up the desirability of agent goals together with the cost of achieving them, weighted by the probability of success. The function provides more than a simple binary decision to aid the choice of actions and helps agents to behave more adequately or even rationally in the sense stated above (Russell & Norvig 2003).

1.2 Motivation

Planning, an approach central to AI research, is substantial for rational agent behavior. It is a method that aids agents in solving complex problems in synthetic and natural environments. Although planning systems are devised for means-end reasoning and are capable to find actions that achieve goals, they are less useful to decide, which goals to pursue (Shut & Wooldridge 2001). BDI systems, on the other hand, are build upon two central ideas. One of them is the reactive planning, comparable with hierarchical planning systems (de Silva & Padgham 2004), the other is goal-deliberation.

It seams reasonable and interesting to combine the strength of flexible means-end reasoning given by AI planners with the timely reactivity and goal deliberation capabilities carried by BDI systems. It is also interesting to analyze suitability of the AI planning approach to BDI agents in real world applications.

The distribution of control and reasoning found in multi-agent systems requires some means for coordination, which may be aided by the planning process. This motivates to treat plans and intentions explicitly, using a representation that may be communicated with other agents. Such representations have been provided by

the planning research field for the purpose of reasoning, manipulation and plan comparison.

This thesis concerns with adaptation of planning techniques to an agent system based on the BDI model of agency. In particular, it describes the approach taken for integration of an *AI* planner into the JADEX(Pokahr, Braubach & Lamersdorf 2003)[1] architecture developed by the Distributed and Information Systems Division at the University of Hamburg. Jadex incorporates many ideas from foregoing BDI-systems, like PRS, JACK™ or JAM[2] and constitutes a JADE agent platform add-on. JADEX has been developed in the JAVA™ programming language.

Further this thesis provides a survey of planning systems and agent systems incorporating planners. Another motivation for this work was to examine the usability of object-oriented descriptions for planning and reasoning. Thus, the diploma project provides a planner that is capable of processing JAVA™-like language containing types, classes, instances, methods and attributes as primary concepts for knowledge representation.

1.3 Approach and Outline

The following chapter introduces basic formalisms concerning AI planning. Among others, a formal definition of planning problems is given. Languages describing the problems are presented together with common planning domains and their properties. The remaining part of the chapter addresses techniques of planning. Partial order planning, hierarchical task networks and deductive planning techniques are presented. Supported by theoretical results with respect to partial order planning, a planner based on these techniques has been designed and implemented. The poor performance of this approach (cf. sec. 5.2) moved the thesis work towards investigation of more practical solutions.

Chapter 3 illustrates the use of planning techniques within concrete systems. In particular, this chapter seeks to investigate recent planning approaches and come behind the secrets of their successful application. Chapter 4 introduces one of the most successful models of agency based on the Theory of Practical Reasoning by Bratman. It concludes with a survey of example BDI systems with JADEX – the basis system for the thesis project. The chapter continues with BDI systems that include an AI planner. The goal of this survey is to examine, what previous approaches have been taken in joining both fields and to investigated if any of these approaches are applicable to JADEX.

Chapter 5 presents the design of the planner and of the way it should be integrated into a BDI system. It accentuates the decisions taken and justifies the choice of planning techniques. The integration with JADEX are described at last. In Chapter 6 two simple planning domains illustrate the use of the planner. The thesis concludes with a summary and outlook in Chapter 7.

[1] `http://vsis-www.informatik.uni-hamburg.de/projects/jadex/`

[2] PRS – Procedural Reasoning System (Georgeff & Lansky 1987); JACK – Jack Intelligent Agents™ (Busetta, Ronnquist, Hodgson & Lucas 1999); JAM – A BDI-theoretic Mobile Agent Architecture (Huber 1999).

Chapter 2

Concepts and Methods of Planning

In order to investigate the strengths and weaknesses of planning methods that could be used in agent systems and applied to multi-agent domains, this chapter introduces fundamental concepts of planning. The applications of planning are presented and formalized under the term of a *planning domain*. Based on these concepts and the understanding of the *planning problem*, different planning techniques are described and differentiated from other techniques found in this research field. The section about *online planning* describes methods applied to problems that must be handled at runtime.

2.1 Primary Concepts

The basis for planning is the representation of the *planning problem*. This section reviews forms of this representation used mainly for classical planning. Given the representation and the description of a domain, upon which the problem was posed, the planner can use some forms of abstraction to focus the processes of planning on important details. Again, one of the central methods for classical planning is *search* described in this section because of its importance. Deduction is yet another method described in one of the following sections. Constraint Satisfaction Problem (CSP) techniques - used for non classical planning - are left unconcerned by this thesis. The last techniques are used mostly for scheduling problems and planning with time constraints. The underling scientific results in respect to CSP are mainly of theoretical nature and thus yield little for the coming chapter, in which direct practical applications are used to further delimit and investigate the appropriate techniques interesting for this thesis.

2.1.1 Planning Problem

The following characterization of the planning problem is based on Ghallab, Nau & P.Traverso (2004)[1]. In order to represent a planning problem one needs at least means to describe states of the world and how these states may change due to agents actions. In a restricted view this can be given by a model of a state-transition system $\Sigma = (S, A, \gamma)$ where S is the set of states, A – the set of actions and $\gamma : S \times A \rightarrow S \cup \{\perp\}$ is the transition function mapping a state and action to another state. $\perp$ is the illegal state being the result of an action not applicable to a foregoing state. The planning problem is given by a triple $\mathcal{P} = (\Sigma, s_0, g)$ where $s_0 \in S$ is the initial state and g is the description of a goal state inducing the set $S_g := \{s \in S \mid s \text{ satisfies } g\}$.

There are at least two representations for planning problems. The first described here is called "classical representation" (Ghallab et al. 2004). It is based on a first-order language $\mathcal{L}$ without function symbols. The states are represented as a set of ground atoms from $\mathcal{L}$. A set of literals g is satisfied in S if there is a substitution σ and every positive and no negative literal from $\sigma(g)$ is in S.

The actions are ground instances of operators defined as a triple $o = (\varsigma, \Pi, \Delta)$ where ς is a signature of this operator containing its name and parameter variables. Π is the set of preconditions required by this operator to be applicable and Δ is the set of effects of this operator. Pi and Δ are sets of literals. Π_a^- is the set of negative preconditions required to be absent from a state if a should be applicable. Π_a^+ are the "positive" preconditions required in a state. Δ_a^- are the negative effects[2] of action a, deleting the specified atoms from a state if applied. Δ_a^+ are positive effects that add atoms to a state.

If action a is applicable in state s, i.e. $\Pi_a^+ \subseteq s \wedge \Pi_a^- \cap s = \emptyset$, the transition function is defined to be:

$$\gamma(s, a) = (s - \Delta_a^-) \cup \Delta_a^+$$

γ maps to an illegal state $\perp$ in the other case. Goals are specified as a set of ground literals.

If problem $\mathcal{P}$ has to be stated to a finite problem solver, care must be taken not to enumerate the states from S as its cardinality grows exponentially with the number of ground atoms in $\mathcal{L}$. The set of all actions A is polynomial in size with respect to the same number of ground atoms. The *statement* of $\mathcal{P}$ can be noted as $P = (O, s_0, g)$ (Ghallab et al. 2004), with O being the set of all operators.

State-variable representation, which uses a functional notation for states, is the second way (mentioned here) to describe a planning problem. A state is represented using state variables of the form $x(v_1, v_2, \ldots, v_k) \in X$ where x is the name of the variable and v_i are objects or variables taking value of an object. For every state

[1]The symbolic notation used here has been simplified and provides a more concise form of expression.

[2]Negative effects are also called the *delete list* and positive effects are called the *add list*

variable there is a corresponding function

$$x : D_1 \times \ldots D_k \times S \rightarrow D_x$$

where $D_i \subseteq D$ are sub domains of this function, being a union of one or more object classes from the whole domain D. Variables denoting predicates can be mapped to a boolean domain, i.e. $p : \ldots \rightarrow \mathcal{B}$. There is no necessity to state functional axioms with this notation as it would be in case of the classical representation.

An operator is a triple $o = (\varsigma, \Pi, \Delta)$ like in the classical representation except for Π being a set of expressions on the state variables and Δ being a set of assignments to state variables. The planning domain is defined as a state-transition system $\Sigma = (S, A, \gamma)$, where:

- S is the set of states defined as assignments $s = \{x \mapsto c \mid x \in X \wedge c \in D_x\}$.
- A is the set of all ground instances of operators.
- $\gamma(s, a) = \{x \mapsto c \mid x \in X\}$ where c is specified by the assignment $(x \leftarrow c) \in \Delta_a$ or by the previous state s if not concerned by action a.

Ghallab et al. (2004) describe the *statement* of a planning problem to be a tuple $P =< O, R, s_0, g >$ where O is the set of operators. R is the set of static relations that do not change with actions. s_0 is the initial state and g is a goal expression on the state variables.

Time representation. Time is the most important resource that should be accounted for while planning. State-transition systems contain a notion of implicit time where actions are related to each other using causal relations based on preconditions and effects. Actions spanning over time are barely concerned by classical planning. Extended models of actions contain delayed effects, invariants that should prevail over time intervals and *joint effects* of more than one action performed at the same time. Long-lasting actions spanning over a time interval may overlap and the domain may be modeled with explicit events occurring on specified times, giving the praise to internal dynamics of the domain. The goals themselves can be stated with *dead lines* to be kept in a valid solution.

Time can be handled *qualitatively* for synchronization purposes between actions of agents and the environment, and time is required *quantitatively* as a resource. The structure of time, given by an ordering relation, can be discrete or continuous, totally ordered or branching. The Point Algebra and the Interval Algebra are two means to represent time in a planning system.

Point Algebra works with instants (time points), which are related to each other using a set of *primitive relations* $P = \{<, =, >\}$ inducing the set of qualitative constraints $R = 2^P$ over them. Except for default set operations like $\cup$ and $\cap$ there is a composition relation defined for the set R. Time points and the corresponding relations constitute a PA network, which is a directed graph $G = (X, C)$ with X being the set of instants and C a set of arcs labeled with a constraint from R.

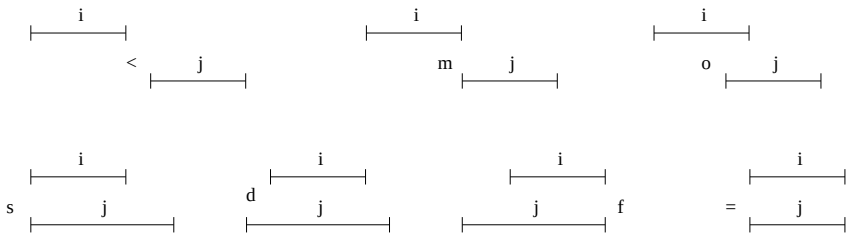

Figure 2.1: The primitive relations of Allan's interval algebra: before, meets, overlaps, starts, during, finishes and equal. The other six relations are pictured by mirroring this ones.

Interval Algebra, due to Allen (1991), works with cohesive time intervals, which are related using thirteen primitive relations (cf. fig. 2.1): before ($<$), meets (m), overlaps (o), starts (s), during (d), finishes (f), after ($>$), met-by (m'), overlapped-by (o'), started-by (s'), includes (d'), finished-by (f'), and equals ($=$). There are 2^{13} qualitative constraints between two intervals, elements of the set $R = 2^P$ where $P = \{<, m, o, s, d, f, =, >, m', o', s', d', f'\}$. As for Point Algebra, $\cup$, $\cap$ and the composition relation are defined for R. An IA network is defined as a directed graph $G = (X, C)$ where X is the set of intervals and C is a set of arcs labeled with an element of R. The consistency check for an IA network is NP-complete as for a general Constraint Satisfaction Problem.

The graph G_{IA} (cf. fig. 2.2) is defined by the thirteen primitive relations connected with an edge, only if one of the relation can be mutated into another one by moving just the starting or ending point of the related intervals. A set of relations from G_{AI} is called convex if for any two elements of this set, all other relations on the path between them are also included. All convex sets define the Convex Interval Algebra IA_c with only 82 qualitative constraints. The consistency problem for IA_c is tractable in polynomial time. Other tractable subsets of IA exists, e.g. IA_p - an Interval Algebra, which includes only constraints that can be expressed as a conjunction of $\{<, =, >, <=, >=, \neq\}$ over the starting and ending points of related intervals.

Temporal Constraint Networks are means to handle time quantitatively. A Temporal Constraint Network is defined as $T =< V, C >$ where V is a set of real valued variables denoting time points and C is a set of unary and binary constraints posed over the variables. Simple constraints have the form $a_i \leq t_i \leq b_i$ or $a_{ij} \leq t_j - t_i \leq b_{ij}$ respectively, with $a_i, b_i, a_{ij}, b_{ij} \in \mathcal{R}$. Generalized constraints are disjunction of the unary and binary ones. On the general constraints set operations and the composition can be defined. For $r = \{I_1, \ldots, I_h\}$, $q = \{J_1, \ldots, J_l\}$:

- $r \cup q = \{I_1, \ldots, I_h, J_1, \ldots, J_l\}$.
- $r \cap q = \{K \mid K = I_i \cap J_j \neq \emptyset\}$ for $I_i \in r$ and $J_j \in q$.
- Composition: $r \bullet q = \{K \mid K = I_i \bullet J_j\}$.

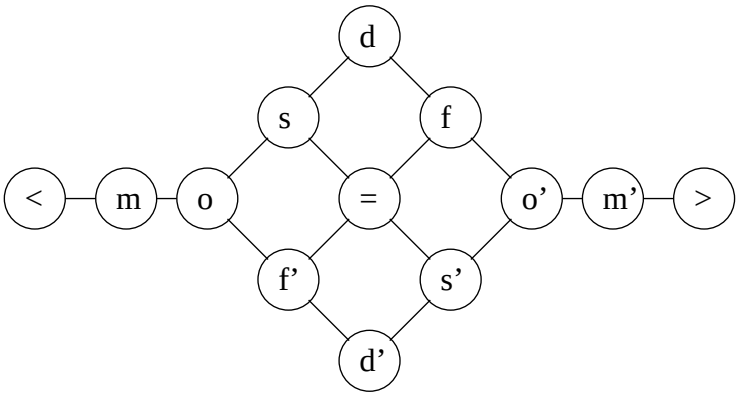

Figure 2.2: A graph G_{AI} showing the neighborhood of primitive relations of interval algebra. Convex constraints contain all relations on the path between two relations included.

With intersection for binary constraints defined by:

$$r_{ij} \cap r'_{ij} = [max(a_{ij}, a'_{ij}), min(b_{ij}, b'_{ij})]$$

and composition by:

$$[a_{ik}, b_{ik}] = [a_{ij}, b_{ij}] \bullet [a_{jk}, b_{jk}] = [a_{ij} + a_{jk}, b_{ij} + b_{jk}]$$

.

Temporal Constraints Networks are mainly used for scheduling and resource planning in domains beyond classical planning using constraint satisfaction algorithms.

2.1.2 Abstraction

One key to cope with complexity in the world is the ability to abstract from its details. Solving problems in real world, humans intuitively use many forms of abstraction. For planning problems there are at least three concepts where abstraction can be applied.

The first concerns with descriptions of the world. *Propositional abstraction* is one way to handle many details in the world by combining them into a single proposition. It is easily achieved by an intensional statement of a predicate describing a fragment of the world state. Another way to abstract from the details is to assign an importance measure to propositions. The planner would plan about its future actions, but regards only propositions above a specified level of importance. With each lower level it would descend in the world abstraction hierarchy using results from the higher level.

Abstraction over the set of available actions reduces the complexity of planning by merging many tasks into one under a common name. An *abstract action* can be acquired by learning out of good action sequences and generalizing these sequences

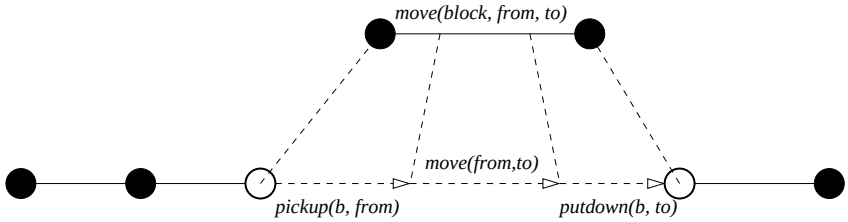

Figure 2.3: The abstract action $move(block, from, to)$ spans over three concrete actions.

to wider problems or it may be prepared by someone else in form of recipes or scripts in advance.

Figure 2.3 shows a part of a script where an object b has to be moved between $from$ and to. This task requires picking up the object b, moving to location specified by to, and dropping that object here. If this is a common task in an environment the *agent* would be advised not to reason about every single action, but to describe the whole task with one symbol. $move(block, from, to)$ summarizes the task of moving *block* and spares the *agent* reasoning about underling actions, while he constructs his plans.

Another way to abstract from the details of actions is to describe the process that takes place while actions are executed. This may be simply represented using *safety* and *liveness* conditions over the loosely composed sequences of actions involved in the process. Informally safety conditions state what states and actions should be avoided where liveness conditions assure what actions or states have to occur in the process again and again. This *process abstraction* may specify meta knowledge about the domain that cannot be related directly to a single action.

2.1.3 Search

Search is one of the fundamental techniques used in artificial intelligence for solving problems. A search algorithm works over a domain called search space. At the beginning a single object from this space is taken as a start. Then using a neighborhood relation on the objects in the space it partially enumerates them, forming a sequence. The search terminates when the sequence - called search solution - includes an object determined to be the goal. Any agent using means end analysis will perform a more or less formalized search in order to achieve its goals and any planning algorithm uses a form of search.

There are several factors determining the choice of a search algorithm. Most important of them is the *branching factor* – b – describing the quantity of objects in the neighborhood relation for a given space and is most important to the space and time complexity of a search problem. Many spaces have an implicit notion of order, i.e they may be searched in forward or backward direction. This fact has to be considered for every search problem as the branching factor may differ greatly between those two directions.

Algorithms that do not use any additional knowledge about the search space

Algorithm	Time	Space
Breadth search evaluates every node at a given distance before going to next level. Useless for larger problems.	$O(b^d)$	$O(b^d)$
Uniform cost search is like breadth search, but the distance between neighbors (*cost* of transformation) may differ.	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^{\lceil C^*/\epsilon \rceil})$
Depth search always takes the first neighbor it finds to proceed. Useless in most spaces.	$O(b^m)$	$O(bm)$
Bounded depth search is like depth search, but fails if no solution can be found under given limit. Very slow for larger problems.	$O(b^l)$	$O(bl)$
Iterative depth search is like bounded depth, but slower. Yields optimal solutions if any.	$O(b^d)$	$O(bd)$
Islands search uses breadth search or iterative depth search to simultaneously analyse the search space from many locations, probably obtained by search in an abstracted space. The paths produced are joined together. Yields performance improvements over the underlying search algorithm.	$O(b^{d/i})$	$O(b^{d/i})$ or $O(bd/i)$

Figure 2.4: Blind search algorithms. b – is the *branching factor*. d – is the depth of first solution found. C^* – is the cost (sum of distances) of the best solution. ϵ – is the minimal distance. m – is the width of the space. l – the limit of bounded depth algorithm to search for a solution. i – number of islands. (cf. Russell & Norvig (2003, pp. 73–81)).

are called *blind* - or *uninformed search algorithms*. A selection is given in Figure 2.4. Solving planning problem, most spaces applicable for search are infinite and blind search has to evaluate an exponential number of solutions (in respect to the size of a valid one) before it will finish. This is why depth search will not even terminate and breadth search will run out of space most of the time.

The informed search algorithms contrast with blind search algorithms, as they utilize a heuristic functions in order to guide their search.

Search control. Using a *heuristic function* is one possibility to control where the search goes. The heuristic search determines its choices on the basis of an evaluation function:

$$f(p, n, \gamma) := g(p) + h(n, \gamma)$$

This function evaluates the partial solution p created up to this point and adds the assumed cost of completing it with the new node n. $g(p)$ is the cost of the current

partial solution p and $h(n, \gamma)$ the heuristic estimate of a solution from node n to the goal γ.

Heuristic functions can be derived from a relaxed problem. It is a subproblem of the original one, constructed by removing some constraints. Given a simple subproblem the algorithm solving it can record the distance of every node to the goal in a database. The main search reads this precomputed distance from the database, or if it has not been computed yet, it invokes a relaxed sub-search on this node. For search problems that can be decomposed into many subproblems, this can be done for every subproblem successively, linearly combining the heuristic cost from each.

Greedy best-first search uses the first node with the best (minimal) value given by the function $f(n, \gamma) := h(n, \gamma)$. If it does not find any solution in a given subspace, it *backtracks* to a former node in the solution. Its search cost in time and space is minimal with good heuristic, but it is not optimal and also incomplete as it gets lost in infinite spaces with rare solutions.

A^* search uses the evaluation function given by $f(\varsigma, n, \gamma) := g(\varsigma, n) + h(n, \gamma)$ where $g(\varsigma, n)$ is the cost of the computed solution to the object n and $h(n, \gamma)$ the estimated cost of the path from the node to the goal. The search keeps all evaluated nodes in an agenda sorted by the computed cost. The node with lowest cost is expanded next. If A* finds a second path to a node on its agenda, the longer path will be replaced with the shorter one and the cost updated respectively.

The star of A* suggests that it is optimal given $h(n, \gamma)$ is an *admissible* heuristic, i.e. a heuristic that never overestimates the cost of a solution. It is also *optimally efficient* as no other algorithm evaluating less nodes is guaranteed to find an optimal solution. On the other hand, for most heuristics, the number of nodes that are evaluated by A^* is exponential in length of the best solution. Practically A^* is incomplete for larger problems as the size of the agenda grows exponentially. Making the agenda limited in size and removing worst cost candidate solutions, as the free memory gets low, makes the algorithm incomplete, but it is a simple alternative to cope with memory problems of A^*.

Knowledge-based control of search is another possibility to reduce the *effective branching factor* in order to speed up the search process. It uses search rules, which depend heavily on the problem domain, even more than the heuristic search control. Modeling this knowledge can take more effort than developing domain heuristics, but it speeds the search greatly as it provides more effective control thereupon.

One possibility to state domain knowledge is to encode it as a set of rules acting on the current state of the search algorithm. Forward-chaining rules, similar to expert systems, may aid the search by interfering with its internal state. Systems utilizing this technique are for example SOAR[3], PRODIGY and the planner UCPOP, described in Chapter 3.

The use of control rules has been seen to be problematic as the user encoding the domain knowledge has to take the workings of the underling search or planning system into account. The rules have been judged to be quite unintuitive (Bacchus

[3]Stands historically for State, Operator And Result (Laird, Newell & Rosenbloom 1987).

& Kabanza 2000). The addition of new rules is difficult as it is the case in many expert systems, so this approach cannot scale well.

Another way to include domain knowledge is to encode it as predicates over partial solution sequences used to prune useless solutions right away. This promising approach using a model checker is described in Section 3.5 together with the $\mathcal{TL}$PLAN algorithm.

The space of planning. The search techniques discussed above have to be applied to a certain search space that must fulfill at least some basic conditions. First, the given space should be discrete or allow to take a discrete snapshot reassembling a specific point in it. There should be a procedure defined on the space points, judging them to be a solution to the search problem, called the *goal function*. For every point in the search space there should be a function mapping it to a finite set of neighbors, called the *neighborhood function*.

For planning problems, the easiest search space is the *state space* corresponding to the classical problem representation using a state-transition system. The search applied to this space is called *forward search*, as it chooses its way through the state-transition graph in a direction similar to the causal or time line. The simplest forward search planning algorithm would take a planning problem statement $P = (O, s_o, g)$ and work its way from the initial state s_0 to a state satisfying the goal g. On its way it would test the preconditions of actions and choose one among some applicable options using a sort of search control. Planning in the state space resembles a controlled simulation. It bears a simple advantage of having at every step almost complete information about the state that can be used to guide the search ahead and apply powerful search control procedures. On the other hand, every search step takes the overhead of copying some or even all of the world representation.

The *problem space* can be used for search if the world cannot be represented succinctly. Planning in this space uses the *backward search*, working in the direction opposite to the time arrow. Given a problem statement $P = (O, s_o, g)$, the goal g describes only a, hopefully small, part of the desired world. This is the initial problem to the search algorithm that works by choosing one of all actions relevant for current goal (i.e. achieving a part of goal requirements) and prepends it to the plan. A new goal is created by composition of the current one and the preconditions of the chosen action:

$$g' \leftarrow \gamma^{-1}(g, a) = (g - \Delta_a^+) \cup \Pi_a$$

where γ^{-1} is called the *regression function* giving this planning approach the name of *regression planning*. The search is continued until the initial state of the planner fulfills the last goal created by the search.

A problem can be decomposed into many subproblems giving rise to different plans, everyone being a partial solution. The idea of combining these plans into a complete solution brings up a new space called the *plan space*. It is different from the previous two, as the points in this space do not describe characteristics of a world, but are partial solutions to the planning problem itself. Every step of the

search is a form of plan refinement, modifying the original plan by removing some sort of plan flaws. The representation of these partial plans is more elaborate than a simple action sequence and provides this way many possibilities for plan improvement. Such representations are Task Networks or Partial Order Plans described in Section 2.3.2. Compared to the state space, plan space planners do not plan with full knowledge of states and have the drawback of not being able to use as powerful search control strategies as in other planning spaces. Summing up, even on classical planning problems, plan-space planners seem to be not competitive enough with state space planners (Ghallab et al. 2004)

2.2 Planning Domains

The environment where each agent acts and which must be considered in the planning process is called *planning domain*. To plan their actions, agents need and require some representation of these domains. Every representation is devised as a sort of abstracting projection taken from the real environment. This section describes the properties of planning domains and corresponding planning problems. The list of attributes here is based on Russell & Norvig (2003) and Ghallab et al. (2004). The rest of this section compares some of the languages used to describe planning domains and problems, and gives some examples commonly used to demonstrate the workings of a planner.

2.2.1 Domain Properties

The properties of a planning domain influence directly the form of modeling used to present the domain to the planner. Each property designates a different aspect of a domain that impact the performance of planning and the choice of possible domain abstractions. Most planners are restricted in the choice of properties a domain model can mirror.

- Observability: Domains can be fully observable. In this case the agent is presented with all information about the domain, it needs to know in order to reason about. If the domain is partially observable, due to limited, noisy or incorrect input, the agent must come up to the lack of his knowledge and possibly incorrect information.

- Contingency: The laws governing changes in the environment can be deterministic. In this case the agent can compute future states, given it knows the current one. In a large domain, with plenty outcomes and limited visibility, the agent is advised to build up a stochastic model of its environment where events are given probabilities in respect to their occurrence. Most classical planning takes an abstract view of a deterministic world.

- Dynamics: The environment of an agent can be static, i.e. the world would wait until the agent finishes its deliberation, or the environment can be dynamic – changing, while the time passes. Environments not only change

with time, but also pose concrete deadlines to the agent, demanding timely responses and emphasis on plans with a strict defined time horizon. Soft deadlines degrade the benefit of an agent while it wastes its time with planning.

- Cardinality: The domains can be divided into finite and infinite domains, having finite or infinite number of states respectively. Continuous domains, in respect to time or object attributes (like speed or weight), are infinite. Classical models for planning assume a finite number of objects and discrete or implicit time.

- Concurrency: The changes caused by an agent and its counterparts including the environment and other agents are perceived to be happening concurrently. The sequential abstraction of those changes, requiring them to follow each other step by step directly impacts the planning algorithm and the way plans are represented. Clearly dynamic domains require model of concurrent changes.

- Agent Counterparts: The number and the kind of agents in the environment determines the way an agent must reason in order to present rational behavior. The domain can be *cooperative* giving place for multi-agent planning and *joint actions*, or the domain may be *competitive* with other agents acting as adversaries. In the last case deterministic models of agency are of less use, because predictability of behavior is a negative factor in the fitness function of an agent. Concurrency issues arise in multi-agent environments requiring for means of synchronization including perception and communication that should be explicitly handled in agent plans.

2.2.2 Languages

STRIPS. The STRIPS notation used in the Stanford Research Institute Problem Solver (Fikes & Nilsson 1971) can be seen as the grand parent of most planning problem description languages. In STRIPS notation an operator is defined by its signature and by its preconditions and effects. The effects can be divided into add effects and delete effects, as the former add propositions to the successive state and the latter remove them. It is STRIPS assumption that all other propositions, not mentioned in the effect list, remain unchanged after the execution of an operator. This assumption is a solution to the frame problem described below.

Planning with the STRIPS notation is PSPACE-complete and even the relaxed problem including only positive effects is NP-hard (Bylander 1992, Bylander 1994).

ADL. The Action Description Language was designed by Pednault (1989) as an extension to the STRIPS notation. In ADL the constants and objects from the domain and variables in action definitions can be typed. This has the benefit of having the planner to investigate less ground instances of actions, as they may only be instantiated with arguments corresponding to the type of parameters. ADL allows for disjunctive and quantified preconditions and conditional effects for actions.

Disjunctive and quantified preconditions are syntactic sugar that can be recompiled into basic STRIPS, given the domain is finite. Universally quantified conditional effects for action can also be recompiled back to STRIPS notation, but generally with exponential space explosion. The closed world assumption on state representation is fallen in ADL. Goals may be quantified and expressions may use a predefined equality predicate.

STRIPS	ADL
Positive literals in states: $at(b, l_1) \wedge clear(b)$.	Positive and negative literals: $\neg at(b, l_2) \wedge \neg on(a, b)$.
Closed World Assumption.	Open World Assumption.
The effect $P \wedge \neg Q$ deletes Literal Q in the successive state and adds P.	P and $\neg Q$ are added, $\neg P$ and Q are deleted.
Effects are conjunctions.	Conditional and universally quantified effects are allowed: $\forall x : (in(x, b) \Rightarrow (\neg at(x, l_1) \wedge at(x, l_2))$.
Untyped, requires type predicates: $Box(x)$.	Objects may have types: $x : Box$.
Goals are conjunctions of positive literals.	Goals may be quantified, include negative literals and disjunctions.

Figure 2.5: STRIPS Notation vs. ADL.

PDDL. The Planning Domain Definition Language is based on ADL and was adapted from the language used for the UCPOP planner for the purpose of the first Planning Competition[4] by a team of researchers under the chair of McDermott & AIPS'98 Commitee (1998). The first version was influenced by notations taken from systems like SIPE-2, PRODIGY, UCPOP and UMCP[5] and looks like the LISP programming language due to the list notation with prefix operators. UMCP provided PDDL with a formalism for representing hierarchical actions with expansion, which was not used in any of the competitions due to large differences between hierarchical planners and was not included in following versions. The PDDL original version had prerequisites for *domain axioms* called later *derived predicates*, safety conditions defined over the domain and "true" negation[6] for evaluation of formulas under the open world assumption.

[4]First International Planning Competition at the Artificial Intelligence Planning and Scheduling Conference 1998: `http://www-2.cs.cmu.edu/~aips98/planning-competition.html`

[5]Erol, Hendler & Nau (1994)

[6]A formula of the form $!P$ is said to be true if it is explicitly said that $!P$. This is opposed to negation by failure where $!P$ is true if there is no statement that P.

The PDDL version 2.1[7], used for the Third Planning Competition, included means for representing domains with durative actions, resources and continuous change. The language was divided into 5 levels of expressive power:

1. Strips
2. Numeric with constraints and resources.
3. Durative actions with discrete models.
4. Actions with continuous models of change.
5. Dynamic models of domain and inherent processes.

Only levels 1 to 3 could be handled by planners in the competition. Numeric expressions have been used in preconditions and effects of actions and are specified in prefix notation. Discrete durative actions can have conditions that hold at the beginning, invariants that hold while executing given action and conditions to be kept at the end. The effects can be applied at the beginning and the end of an action. Also, with PDDL 2.1 came the notion of plan quality metrics, specified using numeric expressions.

PDDL 2.2 is the last version used for the Fourth International Planning Competition. Edelkamp & Hoffmann (2004) kept the first three levels from PDDL 2.1 and supplemented the language with *derived predicates* and *timed initial intervals*. The first extension is taken over directly from the first version of PDDL and allows to define predicates over the domain description that cannot be modified by the effects of an action, but their value can be derived from basic predicates inherent to the domain. The second extension is a restricted form of modeling dynamic changes in the environment. It allows to induce effects on specified instants in the time.

2.2.3 Example Domains

The domains presented here are taken from the International Planning competition 2002[8]. They have been used to test the planners at the competition and are therefore typical planning domains. Further extensions to these domains exist. I.e. the *Settlers of Catan* game scenario has been used to illustrate reasoning and planning by Seegert (2004) in a multi-agent domain.

Depots is a domain build on top of the blocks-world domain and the logistic domain. Crates (blocks) are driven with trucks from a palette to an another one. They are stacked using some hoists. In the numeric version the trucks and hoist consume energy. Trucks have a maximum load capacity and crates have weights. In the timed version durations depend on distances between locations and the speed of trucks. Loading and unloading times depend on the weight of crates and the power of hoists.

[7] Fox & Long (2001)

[8] `http://planning.cis.strath.ac.uk/competition/`

Zeno-Travel concerns transporting people using planes where each plane can move slow or fast (if the number of transported people is less than a threshold). Depending from the mode of transportation planes consume less or more energy and the flight takes more or less time.

Satellite domain. The domain includes an observation task that requires to coordinate multiple satellites to perform observation on different moving objects. The satellites have a restrained energy source where energy is consumed with each action. There is a limited data storage for each satellite, so the number of observations performed by one satellite is finite. The times taking to adjust an satellite for an observation and to calibrate its instruments differ from target to target and from satellite to satellite. The interesting point is that this domain does not have declarative goals and the plans have to be created on the basis of a performance metric.

Settlers This is a domain used to demonstrate planning with resources. Different resources are present in the domain. To mine or harvest the resources it is required to build tools, vehicles and buildings that aid in this process. All of this artifacts take time, labor and use up resources so they can be produced. This is an interesting domain with respect to planning as it grows in number of objects that must be taken into account by the planner.

2.3 Planning Approaches

Former sections concerned planning domains and problems. This section and the following will give an outline over the methods used for planning. Beginning with the *goal stack*, a very simple approach used to solve problems will be shown. Further some elaborated methods will be presented, which have been developed by the planning research community.

2.3.1 Goal Stack

As the name implies the planning problem is posed by pushing goals on the stack. The planner proceeds by taking one goal from the top of stack and trying any actions solving this goal. If the action selected has preconditions not fulfilled by the current state, they are added to the top of the stack and the algorithm restarts with new goals first. Typical planners using this approach are GPS[9] and STRIPS[10] described in Section 3.1.

Goal stack planning has the advantage that goals can be easily added and removed from the stack at runtime and plans are represented in a simple linear order. This method is an efficient option for an agent system to implement an online planning algorithm. Multiple stacks can take care of multiple independent goals, with a scheduler arbitrating in between. The limitations of goal stack lie

[9] General Problem Solver (Newell & Simon 1963).
[10] STanford Research Institute Problem Solver (Fikes & Nilsson 1971).

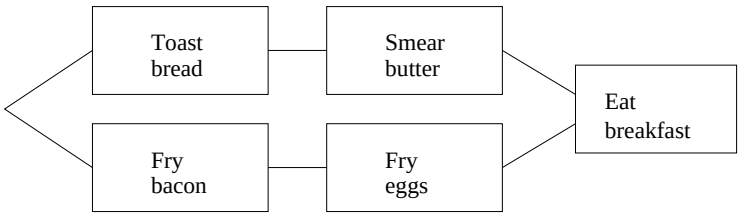

Figure 2.6: A simple partial order plan for breakfast.

in its incapability of handling multiple conjunctive goals in an effective way. To illustrate this, lets take a goal of $A \wedge B$. The stack, represented as a linked list, is shown below:

$$\bot \leftarrow A \leftarrow B$$

Where $\bot$ indicates the bottom of the stack. Now the planner chooses an action, which achieves B and soon after this action has been performed the stack becomes:

$$\bot \leftarrow A$$

At this point the planner has forgotten that it ever had pursued B. Any means of achieving A with side effects making proposition B false will be contrary to the previous conjunctive goal. So after achieving A it is not necessary true that $A \wedge B$. Additionally, because of the fixed order of goals on the stack, the plans and effects of these planners are very sensitive to the order of goals stated in the problem description.

2.3.2 Non-linear

The term *non-linear planning* refers here to the underling representation of plans. As opposed to most state space approaches non-linear planners use partially ordered plans and work in plan search space. This section will concern planning using partially ordered plans also called *task networks*. Interesting is the fact that task networks aroused together with the HTN planning paradigm (cf. sec. 2.3.3) and represent only a simplified or introductory version of it.

Figure 2.6 depicts a simple partial order plan for having breakfast. It consist of five actions that are partially ordered like in the case of toast and butter. The plan has open preconditions requiring to have a bread, bacon, eggs, stove and a toaster too. There would be a causal link between actions of frying eggs and eating them because the former provides fried eggs that can be consumed during the latter.

A partial order plan is a tuple $\phi = (A, O, C, \Pi)$, with A being the set of action instances. O defines the partial orderings over elements from A. C is the set of *causal links* between actions, each being a triple $c_{ij} = (a_i, p, a_j)$. Causal links represent direct dependence between two action instances ($a_i \xrightarrow{p} a_j$) and state the reason for the dependence in form of a proposition p, which truth is assured by the execution of a_i and is required by the action a_j. The last element of ϕ is the set of open preconditions Π containing all preconditions of actions from the partial plan, not covered yet by the initial state or other actions. In a valid complete plan Π is

empty.

The planning starts with an empty plan, containing no other actions, but the fictive ones called *start*, providing all propositions true in the initial state, and *finish* containing preconditions that must be valid in the final state for the plan to achieve its goal. The planner proceeds to complete the plan by removing flaws. It uses resolver procedures performing refinement operations. The refinements, used to complete or repair current imperfect solution, do not pursue particular goals, one at a time – like in goal stack planning or some of the state space approaches, but are chosen in order to find a solution satisfying all of the goals. The planner can choose some of following plan refinement operations:

- Actions may be added to resolve open preconditions.
- Ordering constraints may be put over the structure of actions so the number of conflicts between actions can be lowered.
- Causal links may also be added to provide support of open preconditions from actions already present in the plan.
- Variables may be bound to constants in order to ground expressions.

The partial order representation gives this way a search space with much more degrees of freedom. The number of ways, in which plans could be modified or repaired is the branching factor in this search and it is almost ever much higher than in other approaches. Without proper control knowledge or effective heuristics, partial order planners are much more prone to fail due to the search space explosion.

Despite the very flexible representation of plans in respect to their execution and the fact of heaving explicit notion of causality, the best partial order planners like UCPOP could not compete with simple state space planning approaches due to the difficulty to adopt useful search control procedures. The following section mentions hierarchical task networks, being an extended version of partial order plans allowing for more successful way to plan in the plan space.

2.3.3 Hierarchical

Hierarchical planning uses action abstraction in order to reduce the complexity of the planning process. Approximation hierarchies, introduced first by Sacerdoti (1974) in the ABSTRIPS planner, are build by abstracting one domain representation into a sequence containing more and more details. The idea of ABSTRIPS was to plan first ignoring most of the precondition and successively refine them.

On the other hand action hierarchies allow to describe complex tasks consisting out of many ones. Abstract actions describe a task in the plan space, bridging over many actions and situations. Their value in search for a valid plan is to inform the planner about a possible sub solution described in this abstract way. The planner does not need to force its way in the search space using small steps and does not need to consider any preconditions and effects of underlying actions first. The search can span over large chunk of search space right away (cf. fig. 2.3). It is similar

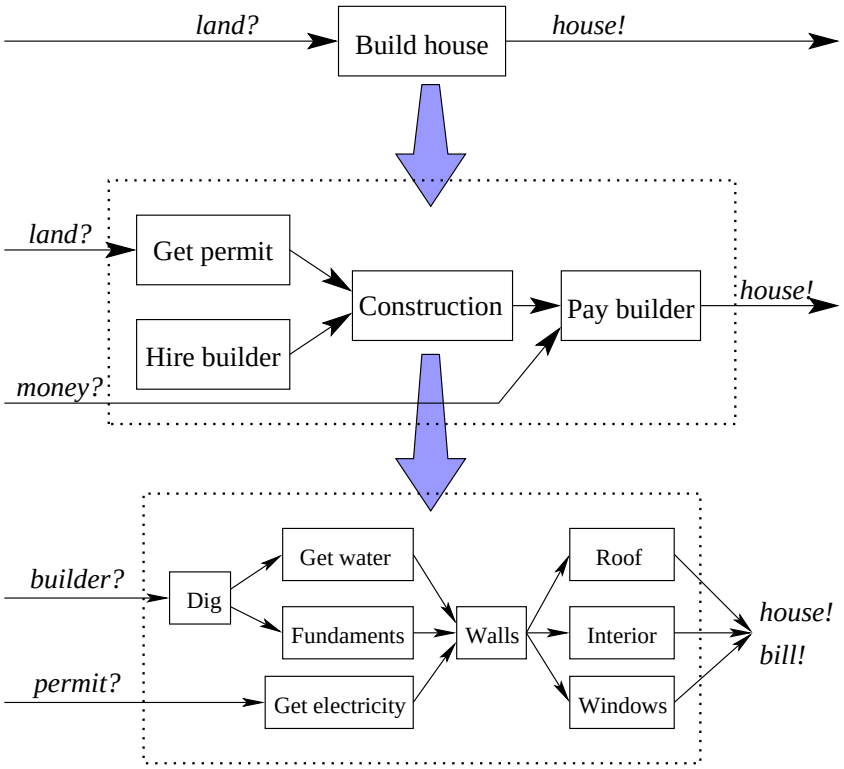

Figure 2.7: The primary high level action *Build house* is decomposed successively into more concrete ones.

to *isle search* where the islands appear in the search space at the joints between decomposed sub actions. This gives great advantages in search speed because almost every search space grows exponentially in the number of action steps.

HTN. Hierarchical Task Networks arouse from the idea of nonlinear planning combined with abstraction hierarchies of actions. A HTN planner works by processing actions on a higher level into a set of actions on a lower level. For every abstract action there is a number of decompositions, which applied, result in partially ordered set of descending actions that have to be integrated into the plan. An abstract action from the current plan stands for different decompositions, represented by partially ordered HTNs with their own open preconditions and effects. The decompositions are commonly stored in a plan library indexed by the abstract action. They differ from inside by the choice of interior actions and from outside by additional preconditions and effects. Figure 2.7 shows a decomposition of an abstract action *buildhouse* with preconditions being a piece of land and the desired effect. It is replaced in the plan by a *task network* composed of actions like *Construction* or *paybuilder*. The consequence of using this decomposition comes in form of an additional precondition, describing the financial resources needed to actually pay the builder. The action *construction* is further decomposed into actions responsible for constructing the house. All the preconditions of the last decomposition are

satisfied by the actions surrounding *construction* at higher level, so there are no new ones, the planner would have to take care of.

HTN hide information from the planner in many ways. The additional preconditions and effects of a decomposition are ignored until the plan is refined. The information, when particular preconditions are required in the decomposition, and where given effects originate are also hidden at the higher level. The preconditions and effects inside the decomposition, together with causal links between the inner actions are also hidden until an action fails and the plan has to be repaired by an online planner.

Many successful planners use HTN techniques for planning. The power of abstraction and elegance in stating additional domain knowledge above the level of operators and prepositions allow HTN planners to compete and win with other planning approaches and even target industrial planning problems. This paper describes in Section 3.3 among others the SHOP planning family, the SIPE-2 and the O-PLAN planners using HTN.

Planning using action decomposition is similar to reasoning with the use of forward chaining rules like: $Cause \Rightarrow Response$, where $Cause$ is the abstract action to be decomposed and the *Response* is the actual decomposition. This aspect of hierarchical planning is particularly interesting for reactive online planners like the Procedural Reasoning System from Georgeff & Lansky (1987) described in Section 4.2.

2.3.4 Deductive

Deductive planning is based on propositional satisfiability algorithms. The planning domain and problem are transformed into a first order logic formula that is fed into a resolver or other theorem proving algorithm. This yields a model, which is consequently decoded into a sequence of actions. Planning using satisfiability techniques is a type of search in the space of proofs. It has been put into focus by Kautz & Selman (1992)

They propose an encoding of a planning problem $\mathcal{P} = (\Sigma, s_o, g)$ to a propositional formula $\phi = \mathcal{E}(\mathcal{P}, n)$ where n is the assumed length of a valid plan. The initial state is fully specified as conjunction of positive and negative literals:

$$\bigwedge_{f \in s_0} f_0 \wedge \bigwedge_{f \notin s_0} \neg f_0$$

The goal also is denoted by a conjunction:

$$\bigwedge_{f \in g^+} f_n \wedge \bigwedge_{f \in g^-} \neg f_n$$

The actions take place at a given step. E.g. $move(b, f, t, 1)$ encodes the action of moving b from f to t at step 1. The preconditions and effects of such an action

are noted by the generic schema:

$$\forall \tau \ . \ a_\tau \Rightarrow \Big(\bigwedge_{p \in \Pi_a} p_\tau \wedge \bigwedge_{e \in \Delta_a} e_{\tau+1} \Big)$$

In order to produce sequential plans the *complete exclusion* axiom for actions must be stated:

$$\forall \tau \ . \ a_\tau \wedge b_\tau \Rightarrow a = b$$

For the satisfiability algorithm to succeed, there is a need not only to model what changes with an action, but also what remains unchanged by it. This concern is famous under the name of *frame problem* being of two kinds. First, for every *fluent* (a proposition that changes with time) and every action the representational frame problem requires to state in form of an axiom if the fluent is affected by the action or not. In classical representation there is an axiom for each fluent and action pair describing the transition of this fluent under the action's occurrence. This representation requires generally $O(|F| \cdot |A|)$ axioms where F is the set of fluents and A the set of ground actions. In the *explanatory* version of the encoding there is an axiom ascribing every change of a fluent to a disjunction of actions actually having an effect on it:

$$\begin{aligned} \forall \tau \ . \ \neg f_\tau \wedge f_{\tau+1} \quad &\Rightarrow \quad \Big(\bigvee\nolimits_{a \in A | f_\tau \in \Delta_a^+} a_\tau \Big) \wedge \\ f_\tau \wedge \neg f_{\tau+1} \quad &\Rightarrow \quad \Big(\bigvee\nolimits_{a \in A | f_\tau \in \Delta_a^-} a_\tau \Big) \end{aligned}$$

Second, the planner must have full representation of all fluents in every state it examines on the way to a solution. The transitional frame problem asks how to represent the change. Should only changed fluents be annotated at a new state or should all information be copied.

It is generally impossible to state all of the preconditions for an action. This is called the *qualification problem* and was stressed particularly in context of deductive planning. It may be illustrated with the simple example of moving a block from a position to another. It is required that the block must be unobstructed and must be located at the start position and the destination has to be free. Additionally the block must be small in size and weight for the agent to pick it up. The agent must be able to use its gripper and have enough power to perform the action. The places must not be separated by an obstacle that would prevent the agent to reach the destination ...

There is another problem dealing with the aspect of implicit effects of an action. It is called *ramification problem* and is best illustrated in the briefcase scenario. When an agent moves a briefcase from one location, all artifacts contained in there must also change their location. The last is an implicit effect as it would not generally be included in the representation of *move*. The location change of these artifacts must therefore be deduced from the effects explicitly stated by the action.

The Situations Calculus. One of the first logical representations used for planning purposes was the situation calculus introduced by McCarthy (1963). Later it was equipped with a full axiom system by McCarthy & Hayes (1969). There is no explicit occurrence of time, which has been replaced by the notion of a situation. The calculus is based on first order logic, with theory spanning over the whole course of actions. The initial situation is called S_0 and all subsequent situation get the name $do(a, s)$ combining the previous situation s and the action a applied to it. All fluents contain the name of the situation where they are assigned a truth value.

Actions are represented in the same language as situations using a possibility axiom: $preconditions \Rightarrow Poss(a, s)$ and an effect axiom: $Poss(a, s) \Rightarrow effects$. Having first order formulas in preconditions and effects, situation calculus allows for more expressive action descriptions than STRIPS or PDDL. Additional to these two axioms situation calculus requires the specification of frame axioms or so called *successor state axioms*:

$$Poss(a, s) \Rightarrow \Big(f(.., do(a, s)) \Leftrightarrow f \in \Delta_a^+ \vee (f(.., s) \wedge f \notin \Delta_a)\Big)$$

The drawbacks of situation calculus are its slight limitation to problems with a single source of change in the world (presumably the agent itself) and the discrete, instantaneous and implicitly serialized notion of actions.

Planners transforming planning problems to propositional formulas and using satisfaction algorithms to derive a plan are for example SATPLAN and BLACKBOX. The latter employed the first stage of GRAPHPLAN planner to derive propositional encoding of the planning problem and solved it using satisfiability techniques (Kautz & Selman 1999).

2.3.5 Scheduling

> "Scheduling addresses the problem of how to perform a given set of actions using a limited number of resources in a limited amount of time." (Ghallab et al. 2004)

Scheduling is not a form of planning, but in most approaches it is a second phase in a decomposed approach to both. The pure planning takes over the role of producing a structure of actions to be performed with explicit causal relations, whereas the schedule phase projects these actions onto the time axis respecting time and resource constraints. Scheduling is for itself a discipline much older than planning and it is rooted in operations research. It can only be briskly mentioned here in respect to domains involving planning.

A scheduling problem is defined by:

- the number and kind of resources,
- a set of actions annotated with time and other resources needed,
- the constraints defined on the resources and
- a utility or cost function the schedule has to optimize.

Contrary to a planning problem there is no explicit initial state and the goal description has been replaced by a cost function. The constants and objects bound in a planning problem to variables in an absolute way are replaced by resources being changed relatively to their previous value. The causal dependencies and preconditions have been replaced by constraints over the structure of actions and the resources. In the solutions to planning problems actions are assumed to be performed in a continuous and non preemptive way, whereas scheduling allows action that can be split over many time periods. Resources may be discrete and continuous. They can be reusable or consumable and actions may be able to *refill* certain resources up to a given maximum level if specified.

Resources are an answer to the problem of planning in domains with a variable number of objects. In classical planning this problem is evaded by defining many *potential objects* with the drawback of having many hypothetical ground actions being symmetric in their kind, i.e. the choice of a potential object of the same kind does not make a difference in a solution. On the other hand, one can represent such objects as resources, making each one anonymous and getting rid of the redundancy in the reasoning.

The scheduling problems are generally solved by encoding it as a constraint network and solving using generic CSP techniques. The constraints posed on the resources are generally quantitative in their type (cf. sec. 2.1.1) and specify deadlines, availability, durations and latency between actions. Scheduling with resource constraints constitutes an NP-hard problem.

Actual planners considering time and resource constrains on the other hand do not use the time-oriented view, as most scheduling techniques, but they revert to the classical state-oriented representation. Although there is much work on planning with time and resources in the plan space, which seems a natural way for this type of planning, the state space approach benefits from the recent advances in this field (Ghallab et al. 2004). Planners using state space approach and integrating time and resource planning capabilities are for example $\mathcal{TL}$PLAN and $\mathcal{TAL}$PLANNER (cf. sec. 3.5). HTN planners like O-PLAN, SIPE-2 and SHOP2 use time windows and resource constraints in their planning process.

2.4 Online Planning

The techniques described earlier apply to offline planning, i.e. the part of agent reasoning that can be performed while the agent is separated temporally from the environment where the plans have to be executed. The particular task of constructing a plan while running in the area of plan application is called *online planning*. There are multiple needs being addressed by online planning. First, under an uncertain environment with intrinsic dynamics there is no guarantee that an offline created plan will succeed. The agent is advised to monitor the success of execution of its plans and on failure it must find alternative means of achieving its goals, presumably by creating a new plan. The other approach takes into account the fact that most agents are already situated and have to act timely to a changing environment. The agents pursue not only declarative goals, but more elaborate forms

of desires present during their whole time line of existence. This sort of persistent *life-time* goals requires an agent design closely coupling the reasoning system, containing and manipulating its plans, with the execution module. The design, allowing for smooth change in goals and plans of an agent, is described under the term of *continuous planning agent* and is based on partial order plan representation.

2.4.1 Plan Monitoring and Replanning

At every moment of its operation the *monitoring agent* checks the environment and estimates the success of outstanding goals and plans it has left to execute. This is done, in order to react to changes in the environment, which could prevent the agent from achieving its goals. The tests performed by the agent can range from simply checking the preconditions of the next action to be performed, which is called *action monitoring* to more elaborate techniques evaluating the course of the whole outstanding plan in order to prove its success (Russell & Norvig 2003). This is facilitated by annotating every plan step with preconditions needed for the given step and for the actions following as well.

When the test reports that the prerequisites required by the following action or whole plan are no longer fulfilled to the expected degree, the agent has the option to create a new plan or to repair the existing one for reuse. The latter works by choosing a hypothetical state in the course of the present but flawed plan, one that most closely resembles the current situation. The plan is truncated up to this state and a new plan constituting a prefix from current state to the hypothetical one is appended in front of the remaining part.

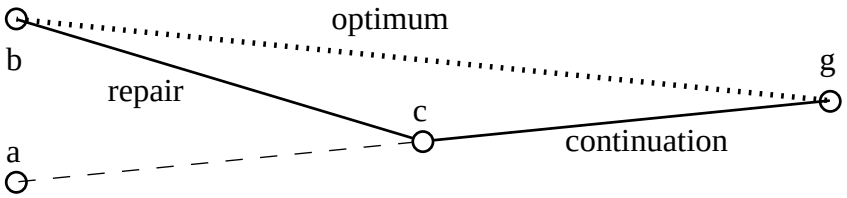

Figure 2.8: A plan repair process constructed an alternative plan in reaction to changed environment. a and b are states, from which the planning process starts and g is some goal state.

The process of repairing a plan can be somewhat costly as the agent is required to find the place where to snip the old one and it works well only if assumptions are taken about the kind of changes in the environment. The repair process for sequential plans can only optimize over the part from current state to the point of cut. Figure 2.8 illustrates this thought. The monitoring module discovered a change in the environment from the assumed state a to the reported by sensors b. The old plan ranging from a to the goal g is not applicable anymore so the agent attempts to repair the plan using a portion called *repair* going from b to a most promising piece c in the original plan. The remaining part of the original plan is called *continuation*. Using the repair process left out the possibility for the agent to find a better plan going directly from b to g.

The replanning technique overlooks the possibility of an agent having wrong assumptions about its own actions and their effects. Possibly due to the qualification problem some actions may fail to provide desired effects in a given environment driving the agent into repeated replanning and redoing the failed action again and again. The same is true in a competitive domain where an adversary would purposely manipulate the environment in order to let agent's plans fail. To address such problem a nondeterministic planning algorithm could be applied and the agent could be made aware of changing cause-effect relations by a more elaborate probabilistic model of its actions amenable to the process of learning.

2.4.2 Continuous Planning

Continuous planning is based on partial order planning. It works with a partial order plan representation, containing at least two pseudo actions. The *current* action is a placeholder for the current state, which is updated by the percepts of the agent. The effects of *current* are the propositions true in the perceived environment of the agent and the action has no preconditions. The *finish* action is an infinitely persistent action containing all goals of the agent as preconditions. The continuous planning algorithm enhances the original partial order plan by refinement or flaw repair procedures[11]:

- Open preconditions are repaired by adding causal links to an existing or a new action.
- Conflicts between causal links ($a_i \xrightarrow{p} a_j$) and actions with effect $\neg p$ are repaired by adding ordering constraints, placing the conflicting action before a_i or after a_j.
- Unsupported preconditions are identified by removing all causal links of the form $current \xrightarrow{p} a_i$ if a given precondition p does not hold anymore in the situation depicted by percepts.
- Redundant causal link can be removed from a path taken from *current* to an action if *current* can provide the precondition of this action and the precondition is not made false on this path.
- Redundant actions are removed if they become irrelevant, i.e. they provide no more causal links.

The execution module of the agent with a continuous planner has the capability to start or stop the planning module. It adds or removes goals, updates the effects of *current*. It schedules an action if all its preconditions are satisfied and the action is ordered before all other actions in the partial plan the planner is working on.

[11] Based on Russell & Norvig (2003)

2.5 Summary

This chapter presented the fundamental principles of AI planning. The purpose was to introduce the reader to this research field and prepare for the coming chapter describing the actual planning systems. The formal definition of planning problems was given on the basis of classical and state-variable representation. Concrete properties of planning domains where delimited and illustrated with common planning application examples. The choice of domain description language determines the complexity of problems that may be stated in this language and the number of features the planner should support in order to handle it.

With the section on planning approaches, some methods have been described that are or have been utilized by planning systems. It has been mentioned that earlier approaches (like the goal stack) may not be effective for even simple problems. Partial Order Planning methods, which occupied the research field for long time, have been dismissed by former much easier state space planning techniques. This is mainly due to the complexity of partial order data structures, reflecting the plans that must be kept and processed yielding complex and less flexible algorithms.

Hierarchical planning is one of most promising planning approaches, mainly in the simple form employed i.e. in the SHOP family of planners and by reactive systems that use hierarchical representation to store their procedural knowledge. The direct application of the HTN representation and planning techniques to procedural reasoning systems may be seen in the CYPRESS architecture that uses a common representation for the HTN planner and the reactive PRS (cf. sec. 4.2.2) component. The RETSINA system takes this thought further. It builds upon a hierarchical planner and modifies it towards a reactive eager commitments strategy (cf. sec. 4.4.3).

Deductive planning and scheduling have been introduced here to complete the chapter and delineate the techniques of planing used in this thesis. Online planning, required by every situated planning system, is presented with two common methods here. Plan monitoring and replanning yields advantages only if planning is more costly then replanning. Continuous planning, in the form given above, has the disadvantage of using partially ordered representation and poor performance. Systems being capable of online planning are SIPE-2 and O-PLAN (cf. ch. 3).

Chapter 3

Planners

This chapter will address the question, what planner technologies are effective enough to be used in the practice. Building on the fundamental concepts and results from the previous chapter, this chapter will cover the techniques on the basis of actually implemented systems that were tested and inspected in various fields of planning.

With the section on classical problem solvers, not only a historical view will be provided, but also first steps taken in the state space and regression planning will be shown. Partial order planners are presented in the section following. UCPOP – one of the most prominent partial order planners – is introduced as a representative for this planning technique. Hierarchical planners illustrate the efficiency and expressiveness of the HTN representation. In particular, the SHOP family of planners utilize a simple but effective approach. Planning graph planners revitalized the field of planning and showed that state space search allows for very competitive planners. This approach will be investigated in various forms. To perform planning efficiently control knowledge may be used. This is investigated by means of the $\mathcal{TL}$PLAN and $\mathcal{TAL}$PLANNER. At last, two artificial intelligence platforms are presented that have been used to investigate planning in a wider context. It is interesting how planning techniques have been composed into these systems and what general view on planning they propose.

3.1 Classical Problem Solvers

Classical problem solvers have been devised as models of reasoning of an unskilled human coming across a new problem. This view required these solvers to be as general as possible and be close to human problem solving techniques. Since then, many classical planners pursued the holy grail of generality even if they lost the ambition to explain the power and efficiency of humans at planning.

GPS. The General Problem Solver stands for a theory of human problem solving implemented in various ways as a simulation program (Newell & Simon 1963). It was one of the first approaches to develop a domain independent means-end reasoner. GPS included as one of the first the idea of cognitive modeling of mental

processes using states and transformation rules. The domains examined with GPS were among others: logic, geometry and chess. Domain models included named operators (as transformations) and objects (e.g logical formulas) for operators to work with. Differences between objects and states have been modeled explicitly as functions and were weighted by the programmer for their difficulty in advance. There were *achieve* and *perform* goals ("transform formula A into formula B") and multiple means to pursue them called *schemes* or *methods*. The search for a valid solution was performed in the problem space and it proceeded in a greedy fashion trying to decompose the main problem into essentially easier subproblems, rejecting all sub-goals more difficult than the original one. The difficulty of goals has been rated by the difference functions. The depth of the goal agenda (realized as goal stack) was limited. The work on GPS was continued by Allen Newell under the SOAR architecture described later.

STRIPS. The Standford Research Institute Problem Solver has been developed as a control algorithm for the Shakey robot at SRI. It used Green's QA3[1] theorem prover and replaced the situation calculus notation with the STRIPS notation. The solver run at that time on a 64kb machine and was able to aid the robot in planning, path finding and object manipulation (Fikes & Nilsson 1971). The algorithm worked in problem space using a goal stack. The approach made STRIPS consider only goals that were immediately preconditions of the last action added to the plan, with an advantage in reducing the search space. Eager commitment to actions with fulfilled preconditions and without a test for validity of the embedding plan also reduced the search space substantially but, similar to the goal stack approach, it made STRIPS incomplete.

The assumption, that all things not mentioned in the change list of an operator remain unchanged, was a simple but very important solution to the frame problem, called since than the STRIPS assumption. But the problem solver was handicapped by its incompleteness that prevented it to find solutions for simple problems like swapping the values of two registers (Ghallab et al. 2004). Due to its limitations the STRIPS algorithm had much less impact on AI planning than the action representation formalism it introduced.

3.2 Partial Order Planners

UCPOP. It is the first partial order planner for a subset of Action Description Language that was proved to be sound and complete (Penberthy & Weld 1992). It was devised to solve the *Yale Stacking Problem*[2] and could stack up to six blocks in the standard blocks-world domain. This was a good result in 1991.

The algorithm of the planner was a nondeterministic partial order planner as described earlier (cf. sec. 2.3.2). It worked with lifted (not ground) instances of operators and goals, so it had the possibility to resolve threats on causal links not

[1] A deductive planner using frame axioms (Green 1969).

[2] A modification of the Sussman Anomaly told to be never solved by a partial order planner (McDermott 1991).

only by ordering the threatening action before (called *promotion*) or after (called *demotion*) the actions constituting the causal link, but also to choose binding constraints on the variables (called *separation*), which would prevent the threat to become effective. Due to the performance concerns of the authors, admitting that planning with UCPOP was not tractable, the algorithm was extended to make use of rule-based search control similar to the PRODIGY design. The control rules have been applied on all nondeterministic choice points in the algorithm.

Further versions of the planner included dynamic object universes, domain axioms and safety constraints, but UCPOP was superseded by the Sensory Graph Plan (SGP), a lisp implementation based on GRAPHPLAN (Weld, Anderson & Smith 1998) praised to be faster, "much, much faster"[3].

RePOP. Revived-POP was designed to show that partial order planning can be made competitive with state space planners. Nguyen & Kambhampati (2001) applied ideas from state space planners, like effective control knowledge, reachability analysis and disjunctive constraints to make a variant of UCPOP algorithm run as fast as GRAPHPLAN.

The REPOP algorithm is based on two steps in a loop. The *flaw selection* and *flaw repair* parts introduce nondeterministic decision points like in an ordinary POP algorithm. Successive plans produced by the flaw repair step are ranked and chosen by the means of a distance-based heuristic estimate derived from a planning graph. The actions are ordered with the help of disjunctions like $S_j < S_i \vee S_k < S_j$, which prevents S_j from destroying the causal link between S_i and S_k. If new ordering constraints are posed over a partial solution, a constraint propagation algorithm prunes all inconsistent orderings and backtracks in case of failure. An additional analysis of indirect conflicts uses two cut-sets for an action S_k enumerating literals that must eventually become true in a state immediately before or after the execution of S_k. A conflict, being a threat on a causal link $S_i \xrightarrow{p} S_k$ is detected if p is mutually exclusive with both cut-sets described above.

3.3 Hierarchical Planners

The main idea of hierarchical planning is to use abstraction in order to create plans on successive detailed levels. With this view most planners including the earliest General Plan Solver may be seen to be hierarchical in nature. Two approaches may be taken to pursue this idea. First one can define importance measures on the propositions in the domain and start to plan ignoring most effects and propositions as described in Section 2.1. This introduces a hierarchy of planning domains and operators, descending from the most abstract ones. The use of these *approximation hierarchies* was first demonstrated in the ABSTRIPS planer by Sacerdoti (1974).

The second hierarchical planning approach utilizes operator abstraction hierarchies. Abstract operators are implemented using different *decomposition methods* as described in Section 2.3.3. Most planners rely on the planning domain designer

[3] `http://www.cs.washington.edu/ai/ucpop.html`

to provide the knowledge for the methods, which makes hierarchical planners of this type dependent on the domain knowledge. As in case of AbStrips it was again Sacerdoti (1975) who introduced nonlinear planning and hierarchies of task networks in his NOAH (Nets of Action Hierarchies) planner.

3.3.1 Sipe-2

The System for Interactive Planning and Execution is the most advanced planner at the SRI International (Wilkins 1999). It was devised as an interactive system for generating, executing and monitoring plans, with efficiency of planning on real world application in focus. The planner can plan automatically or under a guidance of a human expert via a graphical application. The version 2 of the system extended the original system with the capability to specify and use partially ordered decompositions of sub-plans (methods), what allowed Sipe-2 to generate optimal solutions for all possible three blocks problems.

Using domain depended heuristics and *planning advises*, made the planner applicable to several realistic problems. Among others: construction, production line scheduling, controlling mobile robots, military campaigns and propositional puzzles[4]. Sipe-2 constitutes the planning component of the Cypress agent platform and other multi-agent architectures at SRI International.

The algorithm applied in this planner differs from standard partial order or HTN ones. It consists of several modules that can be applied on arbitrary time points in the planning process:

- *Plan critics* are devised to find flaws and constraint violations in the plan.
- *Solvers* are chosen by critics in order to repair the plan. They are not mere flow repair procedures, but can trigger the replanning of actions in the plan or they can remove invalid or conflicting sub-plans from the current one.
- *Interpreter* is called after a new action has been added in order to update the goals and conditions stored in the plan.
- *Monitoring and replanning* is a module that supervises the execution of a plan and reacts to unexpected events in the world.
- *Causal theory* is specified by the designer of the planning domain and describes in a separate way the effects of operators and of external events that may occur at the time of execution. This module is used by the interpreter and by the monitoring and replanning module to insert deduced effects into the plan.

All of the modules can be specified or extended by the user of the system. The plans are generated at different levels of abstraction and the planner maintains a tree of such plans. The replanning process simply changes the world representation and goals of the planner and restarts planning with old plans still stored in the planner. Sipe-2 is implemented in Common Lisp and supports the ACT[5] formalism for the specification of operators as described in Section 3.6.

[4] `http://www.ai.sri.com/~sipe/` – for more Sipe-2 applications.
[5] Myers & Wilkins (1997)

3.3.2 O-Plan

O-Plan is an AI planner architecture extended in a modular way with capabilities to command, control and schedule activities. It aroused from NONLIN, one of the first hierarchical planners and extended the previous approach in various ways to make the planner plan and act in its environment. One central difference to previous planning systems was an agenda of *issues* to collect all *to-do* actions for other components called *Knowledge Sources*. O-Plan can be viewed not only as a planning system but as an agent architecture for situated agents. The version 2 extended O-Plan with a multi-agent approach to planning, scheduling and control where several agents were responsible for processing tasks at progressively detailed levels.

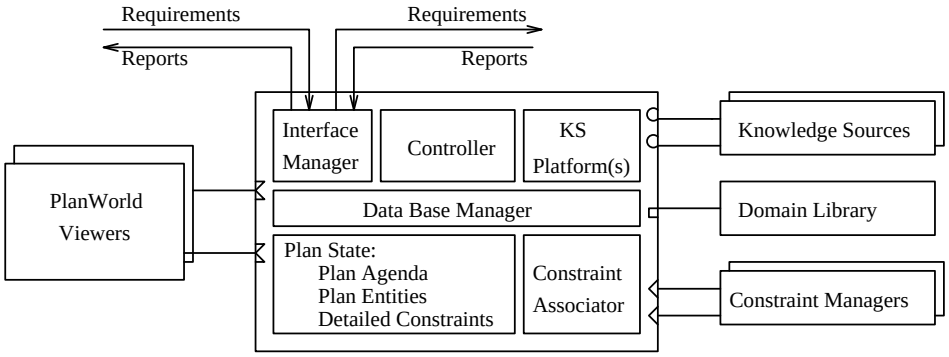

Figure 3.1: O-Plan agent architecture (Tate, Trabble & Dalton 1996, fig. 2)

Figure 3.1 illustrates O-Plan architecture. It is devised to be modular and easily adaptable to new areas of application. Tate et al. (1996) describe the individual modules as follows:

- *Interface Manager* is responsible for communication with the environments and other agents. It collects external events and posts them to the agenda.
- *Controller* assigns issues to responsible Knowledge Sources for processing.
- *KS Platforms* are used to run the Knowledge Sources.
- *Data Base Manager* provides services to other components of the system and keeps the internal state of the agent.
- *Constraint Associator* is an interface to *Constraint Managers* and provides services to them in respect to the Plan State.
- *Plan State* is the internal state of the agent containing all issues, plans and intended actions of the agent. In principle it is a set of (temporal, variable or resource) constraints maintained by the DB Manager. The issues are pending constraints stored in the plan agenda. The plan entities are sets of constraints on actions of the planner. Other constraints include orderings, variable constraints, pre- and postconditions, resources, authority- and spatial constraints.

The customization of an O-Plan agent is done in Common Lisp and O-Plan *Task Formalism* by extending the architecture with domain dependent modules hooked into interfaces present in the system for this purpose:

- *Viewers* are responsible for depicting the situation of an agent and for presenting it to the user controlling the agent. There are plan viewers illustrating the plans constructed and world viewers showing the world in graphical form (e.g. as a land map).

- *Knowledge Sources* are responsible as described earlier for the resolution of planning issues and make heavy use of the domain knowledge for heuristic purpose.

- The *Domain Library* contains a static model of the planning domain and operators available to the planner.

- *Constraint Managers* are used to manage the constraints in the plan and are responsible for making the plans useful for problems in the planning domain.

O-Plan has been applied to many fields of planning including domains of construction, satellite missions, military campaigns or logistic. It was used for production scheduling at Hitachi involving over 300 products, 35 assembly lines and more than 2000 operations. It could construct 30-day schedules with more than a million of individual actions. OptimumAIV is a planning and scheduling system based on O-Plan applied by the European Space Agency (ESA) in order to aid assembly, integration and verification task concerning the production of Ariane IV rocket's equipment bays (Arentoft, Fuchs, Parrod, Gasquet, Stader, Stokes & Vadon 1992).

3.3.3 Shop, JShop, Shop2

Shop stands for Simple Ordered Hierarchical Planner and as the name reveals it is based on a simple domain independent planning algorithm, which uses hierarchical tasks representation to encode the domain dependent knowledge. Nau, Cao, Lotem & Muñoz-Avila (1999) implemented a total order forward search strategy guided by ordered task decomposition and obtained a quite competitive planner (Long & Fox 2002) useful for real world applications (Nau et al. 2004).

The ordered decomposition of tasks reduces the interferences among actions to be resolved. It also gives to the planner the ability to plan with a full state representation at each planning step, allowing for a richer representation of operators and methods, i.e. using numeric expressions over the state or even access to external sources of information (Nau et al. 1999).

The domain knowledge representation for a Shop planner consists of *axioms*, *methods* and *operators*. The following axiom, represented as a horn clause coded in Lisp specific notation, says that for a taxi the agent needs at least $1.5\$ + distance * 1\frac{\$}{km}$.

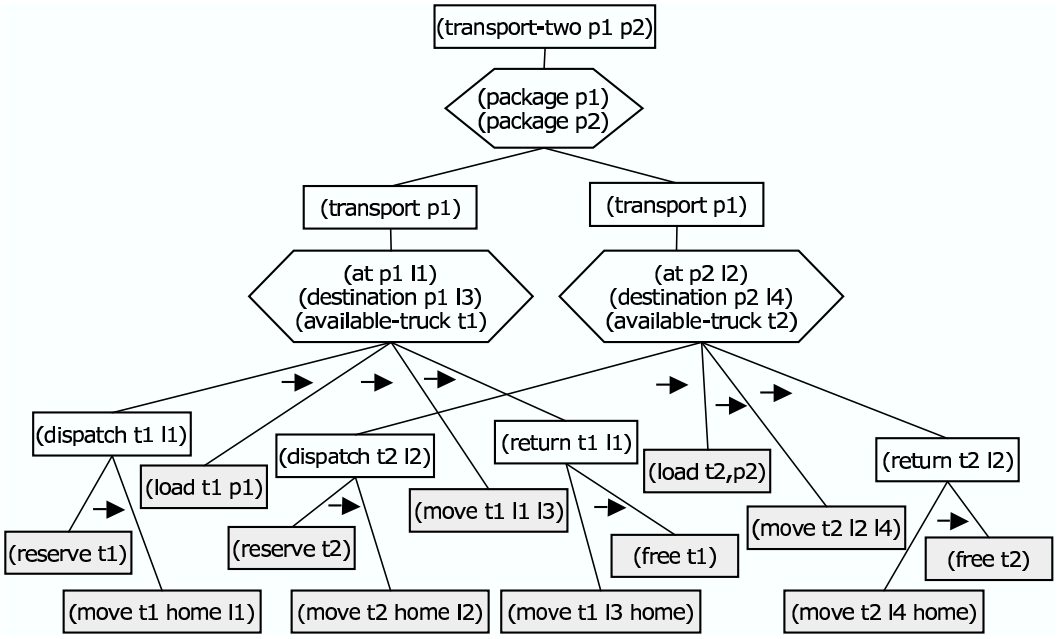

Figure 3.2: A hierarchical decomposition for the task `(transport-two p1 p2)` (Nau et al. 2003, fig. 2). The arrows represent partial orderings between branches.

```
(:- (has-cash4taxi ?d)
    ((has-cash ?c)
     (eval (>= ?c (+ 1.5 ?d)))))
```

An *operator*, similar to the STRIPS operator, is an expression of the form `(:operator H D A)`, where H – the *head* is a primitive task described in LISP, D is the delete list of propositions, and A is the add list. For example the operator below, taken from the blocks-world domain, puts a block on table after the agent was holding it in a hand.

```
(:operator (!putdown ?block)
   ((holding ?block))
   ((ontable ?block) (handempty)))
```

At last the *methods* represent task decompositions. A method description has the form `(:method` H C_1 T_1 $\cdots$ C_n T_n`)`. With H as the method head, denoting the abstract action for which this decomposition applies. The terms C_i denote the conditions to be true for a specific decomposition T_i. If C_j fires all following decompositions ($> j$) will not be considered. The two methods below describe a way of making a block $?y$ clear.

```
(:method (make-clear ?y) ((clear ?y)) nil)

(:method (make-clear ?y)
   ((on ?x ?y))
   ((make-clear ?x)
    (!unstack ?x ?y)
    (!put-down ?x)))
```

Whereas JSHOP is an implementation of SHOP in the JAVA™ language, SHOP2 extends it with several features. First SHOP2 plans with partially ordered subtasks, which allows for more intuitive domain knowledge representation (Nau et al. 2003). Other features concern the extension to ADL and PDDL, and to temporal planning domains.

To represent partially ordered subtasks SHOP2 introduces the `:unordered` keyword in the method representation, as shown for the method `transport-two`.

```
(:method (transport-two ?x ?y)
   (and (package ?x) (package ?y))
   (:unordered
      (transport ?x)
      (transport ?y)))

(:method (transport ?p)
    (and (at ?p ?x)
         (destination ?p ?d)
         (available-truck ?t))
    (:ordered
       (dispatch ?t ?x)
       (load ?t ?p)
       (move ?t ?x ?y)
       (return ?t ?x)))
```

The planner inserts no ordering constraints for the two `transport` tasks (and between actions corresponding to them) into the plan. Figure 3.2 shows a representation of a plan created by the SHOP2 planner for the task of transporting two packages. The planner used the methods `transport-two` and `transport` for the decomposition. A combination of nested `:ordered` and `unordered` keywords allow for specification of more advanced partial orderings. However it is not sufficient for arbitrary partial orderings.

3.4 Planning Graph Planners

All the planners in this section use a partial representation for the (collapsed) search space in the form of a direct leveled graph called *planning graph*. This structure is very useful to provide heuristics and analysis of reachable states. It was applied in many form and complemented with different other techniques.

3.4.1 GraphPlan

GraphPlan has been developed by Blum & Furst (1997). Their motivation was to provide a new way of solving planning problems that was somewhat more efficient than the state of art planners like Ucpop (see above) or Prodigy (cf. sec. 3.6.2). Using ideas from dynamic programming and a structure called *planning graph* they managed to increase the speed of planning substantially (Blum & Furst 1997). The original algorithm is confined to Strips planning domains only. It has a sound, complete algorithm that guarantees to find the shortest plan possible for the planning problem and terminates on unsolvable problems.

The *planning graph* is a directed, leveled graph. It is created in forward manner, step by step, starting from the initial conditions and it guides letter the search for a valid plan. Although it is created in the state space, using discrete time steps, the planning graph is not a state graph, as it collapses the computed states at a given step into one *propositional level*, which contains all at this level achievable propositions. The diversity of valid propositional sets at a given time point is encoded by mutual exclusion (mutex) links between propositions, which have been determined to be exclusive. This exclusion relation is of a binary nature and introduces only simple cuts to the power set of all propositions. So it cannot catch all constraints posed on the planning problem, but can be used as a relative good heuristic.

The propositional levels are interleaved with *action levels*. An action level describes all possible actions at this time point and all incoming (precondition) links are from the preceding propositional level, whereas all outgoing (effect) links go to the succeeding propositional level. The effect links divide into *add links* and *remove links* reflecting the Strips notation. The interference between actions is expressed by exclusion links. Two actions interfere if their effects cannot be achieved at the same time – e.g. action A deletes a proposition, while action B tries to achieve it – or one of the actions deletes the precondition of another. Also, two actions exclude each other if their required preconditions are marked mutually exclusive. The exclusivity relation propagates forward in the graph level by level. Two propositions are marked exclusive if all means of achieving first one are exclusive to all means of achieving the other.

Figure 3.3 illustrates a part of the planing graph created for a rocket domain problem. The first level describes the initial state. Its successor is a level of actions. Straight lines represent preconditions and add effects, dashed lines are delete effects and dots are empty actions (`NoOps`). Given a problem description, including objects o, initial propositions p, actions a with at most k parameters and at most l propositions in the add list, the size of a planning graph is of $O(tp + talo^k)$ for time step t. Since k is constant the size is polynomial in a, l, o, p, t. (Blum & Furst 1997). Despite this optimistic result many problems become unsolvable by Graphplan if the number of objects in the domain becomes large and there are many irrelevant object properties and operators.

When the graph building procedure finds a propositional level, including all goal propositions in non-exclusive relation, a backtracking search over the states is performed from the last level to the initial one. The result of this search is a

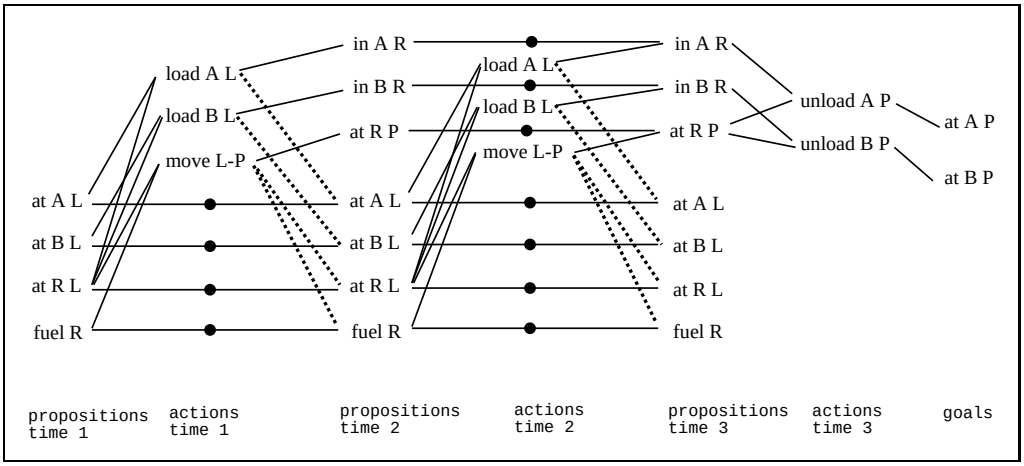

Figure 3.3: Planning graph for a rocket domain. Dashed lines represent delete effect. Black dots represent empty operations (Blum & Furst 1997, fig. 2).

valid plan. If it fails to find one, the algorithm continues with building the planning graph. To achieve given set of propositions Graphplan chooses a minimal set of non exclusive actions to achieve it and moves to a preceding propositional level where preconditions of these chosen actions become the goals. If a set of goals cannot be proved achievable at a time point, it is "memoized" together with the time point. Successive backtracking steps coming over this set of goals will fail right away for any time point less or equal.

The Graphplan terminates after the created planning graph has *leveled of* i.e. no new prepositions are added at a new propositional level and the exclusivity relation remains equal. Additionally the sets of goals memorized as being unachievable do not change.

3.4.2 IPP

Interference Progression Planner has been developed as an extension of the Graphplan algorithm to planning domains contained in a subset of ADL (cf. sec. 2.2.2). It is augmented by the `RIFO`-algorithm to find the set of relevant propositions and operator to a given planning problem. And it uses the `UBTree`-algorithm to aid "memoization" of goal sets proved unsolvable. The subset of ADL implemented in IPP allows it to handle conditional and universal effects and more complex preconditions of actions as opposed to the original Graphplan (Koehler, Nebel & Hoffmann 1997). The operator *move–briefcase* is shown below:

signature: move-briefcase(Briefcase b, Location l_1, Location l_2)
precond.: $at(b, l_1)$
effects: $\neg at(b, l_1)$, $at(b, l_2)$, $\forall x : (in(x, b) \Rightarrow (\neg at(x, l_1) \wedge at(x, l_2))$

It contains a universal and conditional effect. Transforming this operator to the STRIPS notation would require to create one operator for every set of objects that

could be carried within the briefcase resulting in an exponential number of operators.

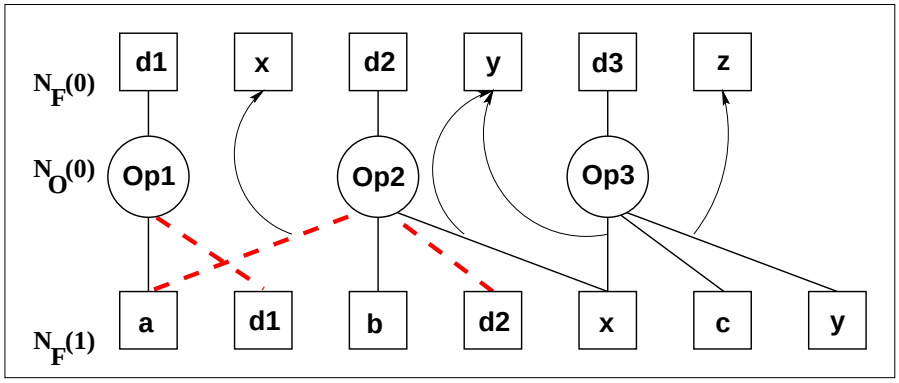

Figure 3.4: Planning Graph with conditional edges represented by arcs (Koehler et al. 1997, fig. 3). $N_F(0)$ is the zeroth level of fluents. $N_O(0)$ is the zeroth level of operators.

The planning graph of IPP is basically the same as in Graphplan, but additionally includes conditional effect edges (cf. fig. 3.4). These edges are augmented by a reference to the set of facts, making the condition of operator's effect true. There can be more than one conditional effect edges joining an operator and a fact, as there can be many conditions allowing for such an effect. A conditional effect is asserted into the graph, when there are facts in the previous propositional level not mutually exclusive that fulfill the condition of this effect. Additionally it is required that facts supporting a conditional effect are not mutually exclusive with facts supporting the particular action.

The plan search algorithm works backwards like in Graphplan, but must cope with conditional effects. The search is performed using two sets of goals G_n and C_n for a given propositional level n. G_n are goals to be fulfilled at the given level like in Graphplan being basically preconditions of following actions. C_n are goals that should not be fulfilled because they would activate conditions in following actions having effects conflicting with the plan up to this point (looking backward).

To process an action level IPPchooses a set of *add edges* fulfilling the goals G_n. This differs from Graphplan where a minimal set of actions is chosen, as in IPP an effect may be achieved by an action in many different ways depending on conditions. The corresponding action is added to the set $\triangle_{n-1}$ of *used actions*. The actions used for G_n are not mutually exclusive, nor do they (unconditionally) delete a goal from G_n or any conditions for the add edges, nor (unconditionally) add a conflict from the set C_n.

A new set of goals G_{n-1} is created from the preconditions of *used* actions conditions for effects included in the set of *add edges* mentioned above. If the set $\triangle_{n-1}$ is minimal (as determined on the basis of required propositions G_{n-1}) IPP calculates the set of conflicting goals C_{n-1}. It includes all propositions from the set C_n if they are not explicitly made false by the effects of $\triangle_{n-1}$. Also includes

all conditions of conditional effects from $\triangle_{n-1}$ that would cause interference or add propositions from C_n. If there are propositions mutually exclusive with any goals of G_{n-1} they can be removed from C_{n-1}. A valid linearization of the set $\triangle_{n-1}$ should not add a condition from C_{n-1} before the action including this precondition (Koehler et al. 1997).

3.4.3 BlackBox

The planner joins the best features of Graphplan and SATPLAN[6]. As the front-end it uses the Graphplan planning-graph creation algorithm and replaces the backward search by a bunch of more powerful satisfiability procedures. The *plan extraction* phase is done by a problem solver in propositional logic after the planning graph has been transformed to a conjunctive normal form (Kautz & Selman 1999).

3.4.4 Fast-Forward

FF is a winner of the AIPS-00[7] planning competition in the *fully automated* category and has been designed by Jörg Hoffmann and Bernhard Nebel with the goal to solve common planning benchmarks in an efficient way. One of the ideas in FF – taken from the HSP system (Bonet, Loerincs & Geffner 1997) – is to use state space heuristics ignoring the delete list of operators, in order to compute an estimate of the solution length counted in the number of actions (Hoffmann & Nebel 2001).

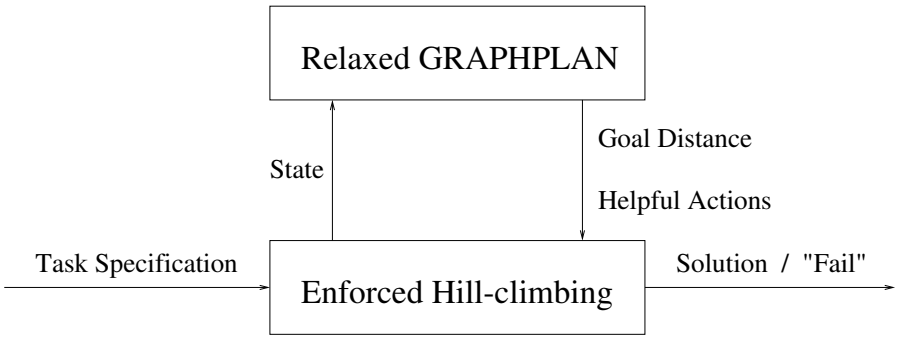

Figure 3.5: FF's architecture with the use of Relaxed GRAPHPLAN exemplified (Hoffmann & Nebel 2001, fig. 1).

For a given search state S, FF computes the solution length heuristic by applying the first stage of the graphplan algorithm to build the planning graph. Then it searches backwards through the planning graph for a relaxed solution, while ignoring the delete effects and conflicts among actions (cf. fig. 3.5). This search chooses `NoOps` first (as in Graphplan), taking actions less "difficult" than others and linearizing actions at a given level in order to come with less preconditions at the earlier level. The difficulty of an action is defined using the minimum precondition level

[6] A satisfaction planner working with logical deduction (Kautz & Selman 1992)
[7] The Fifth International Conference on Artificial Intelligence Planning & Scheduling

heuristic (N_l^f is the propositional level at time point l):

$$h_{dif}(o) := \sum_{p \in prec(o)} min\{l \mid p \in N_l^f\}$$

The size of this relaxed solution $O = \{o_1, o_2, \ldots, o_n\}$ is used as an estimate for the solution length (being the distance to the goal). To reduce the number of times when relaxed GRAPHPLAN is called to evaluate a state, FF uses *enforced hill-climbing*. This technique assumes the search space to be simple in its structure. When the search comes into a state where no other successor state is better it performs an exhaustive breadth first search until it finds one. This form of search performs well, when the plateaus and local minima are small. As the algorithm is *greedy* in nature, it becomes uncomplete when the planning problem includes dead ends. In this case FF switches to a "complete" *greedy best-first*[8] search algorithm.

FF uses the *helpful actions* heuristic to compute successive states for its search. It is extracted from the relaxed planning graph and defined by following:

$$H(S) := \{o \mid prec(o) \subseteq S, add(o) \cap G_1(S) \neq \emptyset\}$$

as the set of operators applicable to S, which add goals to a successive state.

The states are pruned by another heuristic called *added goal deletion* heuristics. If a state S' has been generated from the current state by an action o adding a goal g and there is a relaxed planning graph that includes an action p, which deletes this goal ($g \in del(p)$) then ignore state S' in the search. Both (*helpful actions* and *added goal deletion*) heuristics make enforced hill-climbing incomplete even on tasks without dead-ends (Hoffmann & Nebel 2001).

3.4.5 LPG

Local search for Planning Graphs has been designed by Gerevini and Serina motivated by the search algorithm Walksat used in the Blackbox planner. LPG has won at the IPC-2002[9] in the *fully automated* category.

LPG works by manipulating a subgraph of the planning graph called *action graph* trying to come up with a version without unsupported goals and mutex relations. It is a form of local search that starts at random initial action graph or an initial graph containing mutual exclusions but no unsupported preconditions. This initial graph is extracted from the planning graph built up to the level where all goal facts appear at the first time. Given an action graph A, it chooses at random a flaw f, then calculates the neighborhood $N(A, f)$ of A by applying flaw repair procedures. From the neighborhood the best candidate is chosen on a heuristic basis. The first flaw type is an unsupported precondition, which can be eliminated by adding a new action with effect supporting this precondition or removing the

[8] Russel and Norvig point out that *greedy best-first* search is neither optimal nor complete(Russell & Norvig 2003, sec. 4.1, p. 97).

[9] The 3^{rd} International Planning Competition 2002 hosted at the Artificial Intelligence Planning and Scheduling Conference (AIPS-02); http://planning.cis.strath.ac.uk/competition/

action that requires this precondition. The second flaw type is a mutex relation, which can be eliminated by removing an action in this mutex relation or by shifting the nodes by one level apart, while expanding the planning graph.

The method from Walksat used to aid the search, is similar to the *Lagrange multipliers* method for solving discrete problems and it uses two heuristics. The first heuristic $h_i(a, A)$ gives the cost of adding an action to an action graph:

$$h_i(a, A) := \lambda_p^a \cdot \underset{f \in pre(a)}{MAX}\, C(f, A) + \lambda_m^a \cdot me(a, A)$$

where λ_p^a is the dynamic coefficient for each action a from the planning graph that weights the maximum of estimated costs $C(f, A)$ of the preconditions of A. λ_m^a, on the other hand, weights the number of mutual exclusions $me(a, A)$ for this action. In order to compute the cost of supporting a precondition $f \in pre(a)$, an action a_f is chosen from the planning graph G using the formula below:

$$a_f := \underset{a' \in \{a'' | f \in eff(a'')\}}{ARGMIN} \{\lambda_p^{a'} \cdot pre(a', A) + \lambda_m^{a'} \cdot me(a', A)\}$$

The cost $C(f, A)$ of a precondition $f \in pre(a)$ is: 0 if it is already supported or $C(f', A)$ if a_f is `NoOp` and f' is the corresponding precondition of a_f. Otherwise it is defined by:

$$C(f, A) := \underset{f' \in pre(a_f)}{MAX}\, C(f', A) + me(a_f, A) + 1$$

The second heuristic $h_r(a, A)$ gives the cost of removing an action from the action graph:

$$h_r(a, A) := \underset{f \in pre(a')}{MAX}\, \lambda_p^{a'} \cdot C(f, A/[a])$$

It is the maximum cost of preconditions f (of any a') that become unsupported, weighted by the corresponding coefficient $\lambda_p^{a'}$.

The λ coefficients are updated whenever the search reaches a local minimum by a small amount of δ^+ or δ^-. If the action a has unsupported preconditions, λ_p^a is updated by $\delta^+ \cdot pre(a, A)$. If it is mutually exclusive to other actions, λ_m^a is updated by $\delta^+ \cdot me(a, A)$. If the action does not violate any constraints the coefficients are decreased by δ^-. In this way the search keeps a memory of actions and violations being hard to solve and guided by this information, it pursues successive problems that seem to be easier.

When, in the course of local search, the flaw chosen to be repaired is a mutex (cf. fig. 3.6(a)), one of the mutual exclusive actions can be postponed (cf. fig. 3.6(b)) or anticipated. If this is not possible due to emerging mutual exclusions or absence of helpful `NoOps`, LPG tries to postpone or anticipate this action in combination with graph expansion. The underlying planning graph is expanded by one level and the action graph by inserting a level of `NoOps` in front of the mutually exclusive actions or behind them. Then one of the actions is moved to the `NoOp` level (cf. fig. 3.6(c)). If this is not possible the action graph is modified by removing one of the conflicting actions.

LPG has provisions for *any time* planning, allowing for constructing valid plans

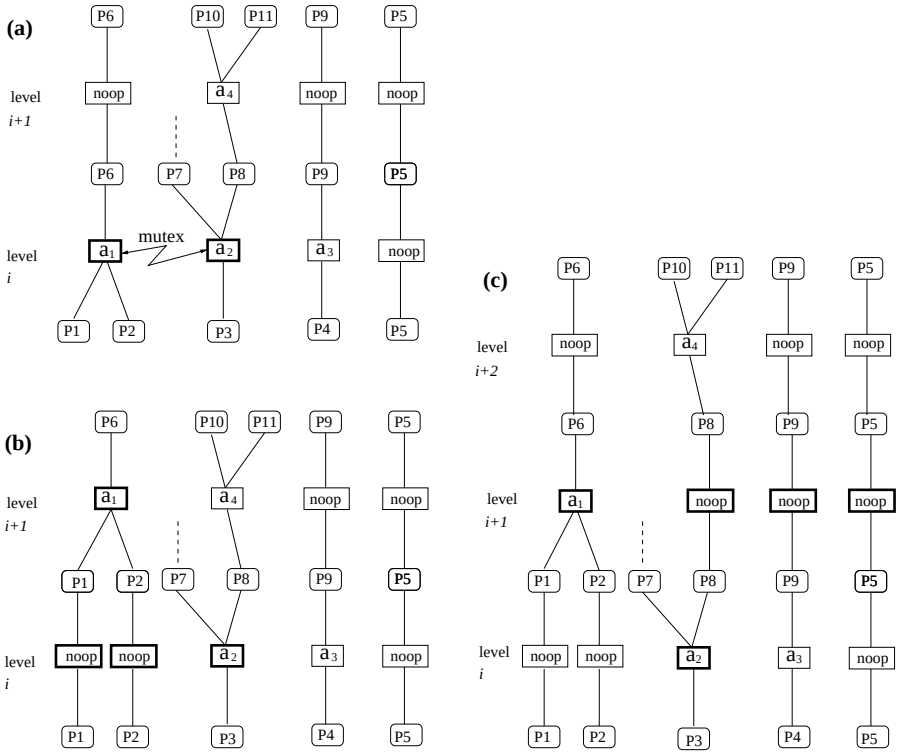

Figure 3.6: Action graph: (a) A constraint violation given by the mutual exclusion of actions a_1 and a_2. (b) A simple modification done by postponing action a_1 to level $i+1$. (c) Postponing a_1 combined with graph expansion. (Gerevini & Serina 2002, fig. 1).

and refining them successively. For this purpose the heuristics described above are enhanced by a weighted term describing plan quality. Every time the search comes up with a valid plan, it can be used for a new search after introducing inconsistencies to it by removing some of its actions (Gerevini & Serina 2002).

3.5 Knowledge Based Planners

In this section two planners are presented that use domain knowledge to control the planning process. The knowledge used by both is state in formulas using temporal logic. These planners are interesting, because they show, how simple search algorithms combined with domain knowledge yield powerful results.

3.5.1 $\mathcal{TL}$PLAN

Temporal logic ($\mathcal{TL}$) planner has been developed by Bacchus & Kabanza (2000) and it is a planner utilizing domain knowledge to make the search for valid plans faster. It was the winner of the *hand tailored* category at the IPC-2002, solving 894 planning problems from the STRIPS, numeric, time, and complex domains. It achieved this with great efficiency being magnitudes faster than competitors in most

cases.

The domain knowledge is encoded in *temporal logic* ($\mathcal{TL}$) formulas describing *good* plan sequences. It is used by the search algorithm to prune in advance many useless plans before their course could be further explored. This differs from the heuristic and other knowledge approaches, which evaluate only the current state of the planning process.

The following example taken from (Bacchus & Kabanza 2000) shows rules specifying good plans within the blocks domain.

$$\begin{aligned} goodtower(x) &= (ontable(x) \wedge \neg\exists[y : GOAL(on(x,y))]) \\ &\vee \exists[y : on(x,y)]\neg GOAL(ontable(x)) \\ &\quad \wedge \neg GOAL(clear(y)) \\ &\quad \wedge goodtower(y) \end{aligned}$$

The predicate $goodtower(x)$ gives the planner an abstracted view on some block structures on the table. Every stack of blocks being in the final goal position is a *goodtower*. The predicate $GOAL$ is a second grade predicate viewed as a modality. It judges atomic propositions and says what goals does the planning algorithm have. This state abstraction can be directly used in the $\mathcal{TL}$ formula:

$$\Box\Big(\forall[x : clear(x)]goodtower(x) \Rightarrow \bigcirc\Big(goodtower(x)\Big)\Big) \tag{3.1}$$

stating that it is always the case ($\Box$) for every good tower x, that it will stay good tower in the next state ($\bigcirc$). The expression $\forall[x : clear(x)]$ is a bounded quantification over all x without blocks on them, as they are the only ones considered for the move action.

One of the advantages of this approach is the scalability achieved by simply adding new formulas in order to prune useless sequences. The next equation states that good towers can only be expanded to good towers:

$$\begin{aligned} &\Box\Big(\forall[x : clear(x)]goodtower(x) \Rightarrow \\ &\quad \bigcirc\Big((\exists[y : clear(y)]on(y,x)) \Rightarrow goodtower(y)\Big)\Big) \end{aligned} \tag{3.2}$$

$\mathcal{TL}$PLAN uses a model checker for $\mathcal{TL}$ under the assumption the the world idles after the prefix end. This assumption is required because models to $\mathcal{TL}$ are generally infinite. Other components of $\mathcal{TL}$PLAN's architecture - apart from the model checker - are:

- *Search engine* a simple forward-chaining search algorithm for the state space,
- *State expander* realizes the neighborhood relation,
- *Goal tester* checks if given goals have been reached,
- *Formula evaluator* used by all other components to evaluate time independent formulas.

The remaining architecture is relative simple as it uses forward chaining search in the state space that is easy to control. Utilizing domain knowledge by encoding it in $\mathcal{TL}$ formulas seams to be strong enough to help $\mathcal{TL}$PLAN to cope with large and difficult problems.

3.5.2 $\mathcal{TAL}$PLANNER

$\mathcal{TAL}$PLANNER is a planning system built on ideas from $\mathcal{TL}$PLAN. It is a forward-chaining planner that relies on domain specific knowledge to prune the search space from invalid or useless solutions. The control knowledge is stated in a *Temporal Action Language* ($\mathcal{TAL}$) being the "... narrative-based non-monotonic linear discrete metric time logic for reasoning about action and change" (Kvarnström & Magnusson 2003). Contrary to most languages used for planning, $\mathcal{TAL}$ allows to state not only static and declarative goals in form of propositions about the final state, but also to describe *safety constraints* and *maintain goals* that must hold through plan execution.

The input to the $\mathcal{TAL}$PLANNER is given in a meta-level macro language $\mathcal{L}$(ND) that is translated to a base level language $\mathcal{L}$(FL). The initial conditions are stated in form of observations at time point 0: `#obs [0] on(block-a, block-b)`. For goals there is a simple goal directive: `#goal on(block-b, block-a)`. The operators take a form similar to the the one in STRIPS notation[10]:

```
#operator move(from, to) :at t
    :precond    [t   ] at(agent, from)
    :effects    [t+3] at(agent, from) := false
                [t+3] at(agent, to) := true
```

This is the internal representation of the $\mathcal{TAL}$PLANNER. In the macro language $\mathcal{L}$(ND) it is represented in the following way:

$$
\begin{aligned}
&\textbf{acs}\ [t, t']\ \texttt{move}(from, to) \rightsquigarrow \\
&\quad t' = t + 3 \wedge \\
&\quad ([t]\texttt{at}(agent, from) \rightarrow \\
&\quad\ R([t+3]\texttt{at}(agent, from) = \texttt{false}) \wedge \\
&\quad\ R([t+3]\texttt{at}(agent, to) = \texttt{true})
\end{aligned}
$$

This translated to the base language $\mathcal{L}$(FL) gives:

[10] Example taken from (Doherty & Kvarnström 2001)

$$
\begin{aligned}
&\forall t, t', from, to. Occurs(\texttt{move}(from, to)) \rightarrow \\
&\quad t' = t + 3 \wedge \\
&\quad (Holds(t, \texttt{at}(agent, from), \texttt{true}) \rightarrow \\
&\quad\ Holds(t + 3, \texttt{at}(agent, from), \texttt{false}) \wedge \\
&\quad\ Occlude(t + 3, \texttt{at}(agent, from)) \wedge \\
&\quad\ Holds(t + 3, \texttt{at}(agent, to), \texttt{true}) \wedge \\
&\quad\ Occlude(t + 3, \texttt{at}(agent, to)))
\end{aligned}
$$

Where *Holds* states at what time point a proposition is true or false and *Occlude* says what propositions are exempt from the inertia and persistence assumptions dealing with the frame problem[11].

$\mathcal{TAL}$Planner has an explicit notion of resources being represented like *fluents*, i.e. propositions that change with time. Resources have a numeric value and can be consumed, produced, borrowed (exclusively or not). Resources can have a minimum and maximum amount allowed.

Search control rules are stated in the $\mathcal{L}$(ND) language and are domain dependent as described above. For example the $\mathcal{TL}$Plan control rule 3.1 can be written in $\mathcal{L}$(ND) this way:

$$
\begin{aligned}
\forall t, x[t \quad\ ]\ & clear(x) \wedge goodtower(x) \rightarrow \\
[t + 1]\ & goodtower(x)
\end{aligned}
\tag{3.3}
$$

which is represented internally in a Scolem form by:

$$
\begin{aligned}
&\#\text{control :name "keep good towers"} \\
&\qquad [\text{t} \quad\ ]\ clear(x) \wedge goodtower(x) \rightarrow \\
&\qquad [\text{t+1}]\ goodtower(x)
\end{aligned}
$$

Before fed into the model checker, the rules are preprocessed and optimized for every operator given in the domain resulting in control rules for every operator acting as a form of precondition. $\mathcal{TAL}$Planner searches the state space using simple progressive depth first search. It differs from $\mathcal{TL}$Plan as it allows for concurrent actions. Given a discrete time line, it inserts as many actions into the plan as possible at a given time point (cf. fig. 3.7), before proceeding to the next point. Contrary to many forward-chaining state space planners, the nodes in the search space are viewed as sequences being a partial logical model for the underlying domain theory. This feature allows to compact plans by the approximated factor of 10 actions to 1 time step - depending on the domain (Kvarnström & Magnusson 2003).

[11] For more precise description of the $\mathcal{TAL}$ logic please refer to the $\mathcal{TAL}$ Specification and Tutorial (Doherty, Gustafsson, Karlsson & Kvarnström 1998)

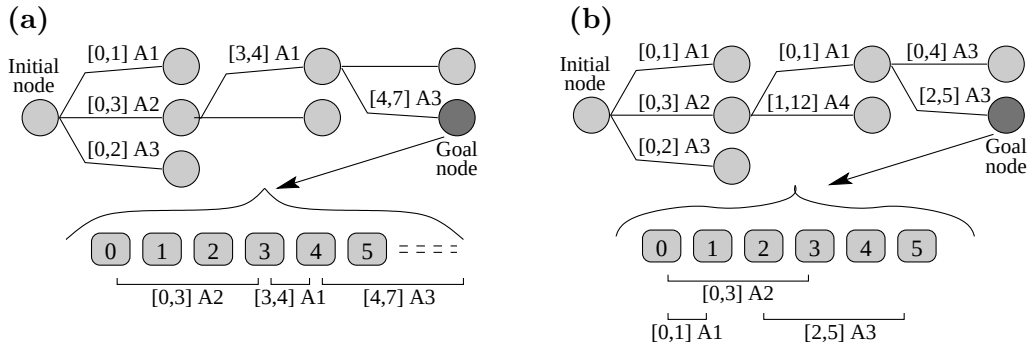

Figure 3.7: Progressive search using (a) sequential, and (b) concurrent actions (Kvarnström & Magnusson 2003, fig. 2).

3.6 Platforms

In this section two architectures for artificial intelligence will be presented. The choice on SOAR and PRODIGY comes from their focus on planning and problem solving. The platforms here are not devised with the theory of agency in mind and do not constitute models directly applied to it. In contrast to the planners described above both try to integrate a broad field of AI techniques into one system.

3.6.1 SOAR

SOAR has been developed since 1983 in mind to research a unified theory of cognition with means of artificial intelligence. It is based on a production system (formerly OPS-5) utilizing the RETE algorithm. Goals of the project were to provide one architecture for systems that exhibit intelligent behavior on a wide range of problems, employ procedural, declarative and episodic knowledge and learn about all aspects of the environment and themselves. The design of SOAR is directed by architectural simplicity. As may be seen in Figure 3.8, there is one source of static long-term knowledge, represented by *production memory*, one source of dynamic short-term knowledge stored in *working memory* and a single mechanism for learning called *chunking* (Laird et al. 1987).

The underlying production system matches conditions of a production rule on the basis of working memory. The rule fires and asserts some new knowledge into the working memory. Given that a rule is no more supported by the working knowledge, its effects may be retracted. Chunking watches the rules and their application and constructs more general rules saving them back into the production memory.

SOAR itself utilizes this subsystem to implement a goal driven problem solver working on the basis of *problem space hypothesis*. The main element of the working memory is a *s*tate to which objects with attributes are connected. The state is manipulated using *o*perators. Because operators are defined by production rules as all static knowledge and there may be many operators applicable to a given state, SOAR employs a working cycle consisting of the stages: perceptual input, operator proposal, operator comparison, selection, application and output.

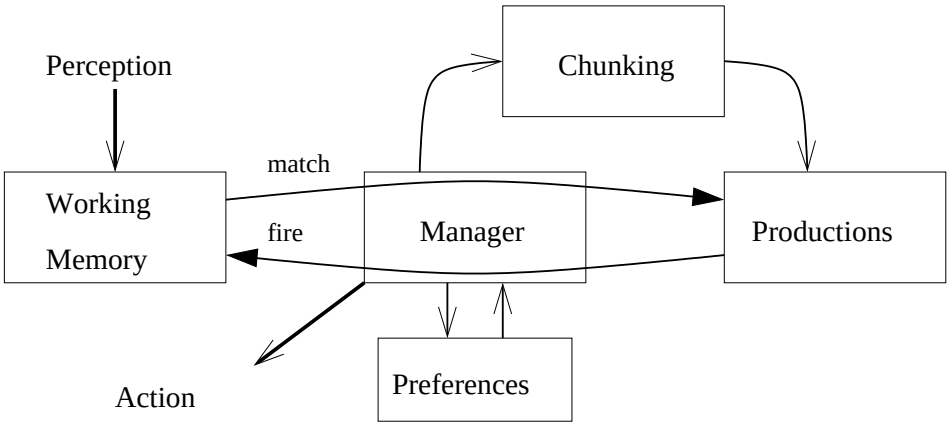

Figure 3.8: SOAR – A simplified modular view. Productions come from the productions memory. Preferences come from the preference memory.

In the operator comparison phase SOAR uses a *preference memory*, being a set of productions, in order to rank the operators so only one of them may be selected. The preferences state if an operator is acceptable, better or worse than any other. They may specify if it is required or prohibited in the current state. But the proposal, comparison or selection phases may not be complete (no-operator) or there may still be some operators to choose among (tie). This is called *impasse* and leads to generation of a sub-state (subgoal). There are other impasses arising if operator A is rated before B and v. v. (conflict impasse) or there are two operators required (constraint-failure impasse). If an operator asserts no additional knowledge into working memory there is a no-change impasse.

The sub-state created has a reference to the super-state and is annotated with the reason of its creation. The SOAR cycle continues to work on the new state. If the impasse can be resolved, the *r*esult of this sub-problem is integrated back into the super-state. For example, if there is a tie impasse, a sub-state is created for each of the operators in choice in order to determine, which one is the best. This may be seen as a form of state space search using forward chaining rules as search control.

As with most production systems the overhead grows polynomial with the amount of knowledge stored in the working memory. This makes SOAR applications not very scalable, especially because the rule matcher considers all states and sub-states at the same time. Additional the *automatic subgoaling hypothesis* employed by SOAR and its implementation using a kind of *goal-stack* approach does not allow SOAR system to react flexible towards problems emerging from different contexts. There is no mechanism to deliberate about goals and problems arise if goals stay in conflict to each other.

3.6.2 PRODIGY

The PRODIGY platform was created as a general purpose planning architecture for testing different learning techniques and their usefulness to aid the performance of

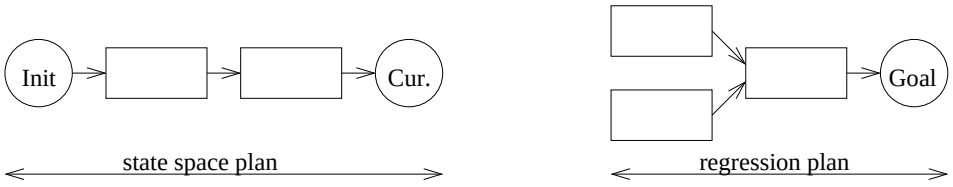

Figure 3.9: PRODIGY partial solution representation (Veloso et al. 1995, fig. 4).

the planning process itself (Carbonell, Knoblock & Minton 1991). The implementation language is Common Lisp, but the planner accepts a reasonable subset of the Action Description Language for the purpose of domain modeling.

The planning by the system is performed on a plan structure presented in Figure 3.9. It consists of a *head part* representing a course of action from the initial state of planning (an abstraction of the environment) to the state called *current* representing the last state reached in the forward directed simulation. The other part, called *tail plan*, is used by a backward chaining planner to reason about causal dependencies among instantiated operators. The general algorithm calls the back chaining part to expand the tail towards the current state. It chooses an open precondition l of an action in the tail that is not yet satisfied by the current state or any other action in the tail preceding this one. It chooses a new operator that achieves l and instantiates it fully before appending it in front of the action mentioned earlier.

Alternatively to the expansion, one of the operators dangling at the tail may be moved to the head. It will be applied to the current state in order to produce a new resulting situation. The choice among this two processes (expansion or application) may be guided by domain dependent control rules. This allows for a forward planning or backward planning alone and for different mixes of the two approaches. Given that the operators are applied only to states with preconditions fulfilled, the algorithm terminates when the head reaches a state that satisfies the goal. Provided that the actions are modeled right this makes for a sound algorithm on account of the forward part only.

The control rules, used by the planner at all of the choice points mentioned above, have the form: `if` *condition* `then` *advice*. Where *condition* is a conjunction of propositions about current state, goal, operator, candidate goals or other information describing the state of search. The advice is of threefold kind. The *select* rules are used first to choose among options. The default, in the absence of other rules, is to select all options. Thereafter *reject* rules are applied to prune the set of selected options. At the end the choices are ranked as specified by *prefer* rules.

Extended learning capabilities have been provided as modules and may be applied to learn control knowledge in order to improve the efficiency of the planning process and the quality of plans generated:

- *EBL* is an explanation-based learning module collecting control knowledge from traces of valid and correct plans.

- *STATIC* analyzes a problem in advance and generates control rules.

- *DYNAMIC* combines virtues of the EBL and STATIC module.
- *ALPINE* produces several levels of abstraction of the original planning problem and tries to solve it in a hierarchical way.
- *ANALOGY* is a case-based approach to guide the planning process.
- *EXPERIMENT* refines the domain models using learning by experimentation.
- *OBSERVE* generates new operators from action traces and refines successive their description.
- *QUALITY* module compares a solution of the planner with a better solution as specified by an expert. The result is a set of rules that should help the planner to output better quality plans.
- *HAMLET* uses a set of training problems and a quality measure to produce control knowledge that would allow for plans with better quality and for less planning failures.

3.7 Summary

Table 3.1 is a comparison of planners put together from many sources[12]. Among others, there are planners that took part in planning competitions. The classification system is an extension of the one presented by Wah & Chen (2003). To clarify their meaning, the table includes following categories describing the internal mechanisms applied by planners and the domains they are able to handle:

- *Heuristic* refers to the heuristic knowledge used by the planners (cf. sec. 2.1.3).
- *Knowledge Based* refers to domain knowledge applied by the planning algorithm in order to ease the search for a solution.
- *Hierarchical* describes the task decomposition approach used by the planner (cf. sec. 2.3.3).
- *Systematic* specifies that the planner performs an almost exhaustive search in its space of planning.
- *Local Search* is complementary to the systematic approach. The planner uses techniques reducing the scope of planning space it has to visit. This includes greedy or depth first searches.
- *Transforming* planners convert the problem representation into one for which there are known solution procedures. This includes for example transforming ADL into first order logic.

[12]Long & Fox (2002), Edelkamp, Hoffmann, Littman & Younes (2004), Wah & Chen (2003), Planning Database: `http://scom.hud.ac.uk/planet/repository/`

- *Fully Automated* says that the planner is capable to plan without any domain dependent knowledge.
- STRIPS is the basic capability of every planner to handle STRIPS domains.
- *Numeric* domains include variables ranging over integral or real numbers.
- *Time* says that the planner can handle durative actions.
- *Continuous Time* domains include time intervals and points described by real numbers.
- *Mixed States* include sets of discrete and continuous items.
- *Contingent* planning regards probabilistic quantities describing actions and the domain.
- *Online* planning is done by systems directly coupled with the environment (cf. sec. 2.4).
- *Anytime* algorithms produce a less valuable solution at first and are able to improve it over the time.

The table contains a graphical summary of the chapter. It exemplifies the reference of methods used to the features of planners presented. It may be seen as an almost chronological path through the development of planning systems. Taken at first, there are simple systematic search planners, which can only handle simple domains. Due to the growing complexity of numeric and temporal domains, almost all new generations of planners employ heuristics and domain knowledge. The explosive size of planning spaces force the designer to go away from systematic exhaustive approaches and concentrate on local solution improvements.

Again, in this table it is clearly shown that approaches as simple as state space search can easily handle domains with continuous time and other quantities. On the other hand, partial order planners give up the simplicity for more degrees of freedom at the planning time. This results in greater complexity of algorithms, less flexibility and, in effect, quite poor results in respect to efficiency of planning, because of inherent planning space explosion.

The choice of a planner for an agent system should be guided by the agent application environment and its properties (cf. sec. 2.2.1). Taking into account that even simplest agent domains become very complex, the choice should fall on planners from the lower part of the table. In particular, planners with online planning and anytime capabilities are interesting for agent systems. SIPE-2 and O-PLAN are already full blown systems with agent-oriented aspects, so their integration into other systems would require to strip a lot out of them. They are useful for agents written in high level languages like LISP, as do LPG and MIPS planners.

In respect to BDI systems described later, the use of a planner to control the course of actions is somewhat contradictory. BDI systems have been designed as an answer to the incapability of former decision theoretic and deductive planning systems to control agents in real time. As described further, it was the outstanding

reactivity and performance that made BDI so successful. Introducing a planner into a BDI system, would only make sense in order to leverage the shortcomings of BDI in respect to reasoning about their actions and future changes of the environment in response to these actions. It should not be made, when the benefits of BDI would be sacrificed. The topic will be further pursued in Section 5.3, concerning with the choice and design of a planner for the JADEX BDI system.

	HEURISTIC	KNOWLEDGE-BASED	HIERARCHICAL	SYSTEMATIC	LOCAL SEARCH	TRANSFORMING	FULLY AUTOMATED	STRIPS	NUMERIC	TIME	CONTINUOS TIME	MIXED STATES	CONTINGENT	ONLINE	ANYTIME
	methods						features								
UCPOP				**S**			**A**	**S**							
GRAPHPLAN				**S**			**A**	**S**							
IPP				**S**			**A**	**S**							
SYSTEM-R[a]		**K**		**S**			**A**	**S**							
HSP[b]	**H**				**L**		**A**	**S**	**N**						
FF	**H**				**L**		**A**	**S**	**N**						
ALTALT[c]	**H**				**L**		**A**	**S**	**N**						
GRT[d]	**H**				**L**		**A**	**S**	**N**						
PRODIGY	**H**	**K**			**L**		**A**	**S**	**N**						
SATPLAN					**L**	**T**	**A**	**S**	**N**						
BLACKBOX	**H**			**S**	**L**	**T**	**A**	**S**	**N**						
SIPE-2	**H**	**K**	**H**	**S**				**S**	**N**	**T**		**X**		**O**	
O-PLAN	**H**	**K**	**H**	**S**				**S**	**N**	**T**		**X**		**O**	
SHOP2			**H**					**S**	**N**	**T**	**C**	**X**			
LPG	**H**			**S**	**L**		**A**	**S**	**N**	**T**	**C**	**X**			**A**
MIPS[e]	**H**			**S**			**A**	**S**	**N**	**T**	**C**	**X**			**A**
$\mathcal{TL}$PLAN		**K**			**L**			**S**	**N**	**T**	**C**	**X**			
$\mathcal{TAL}$PLANNER		**K**			**L**			**S**	**N**	**T**	**C**	**X**	**C**		

Table 3.1: Classification of planning systems.

[a] A STRIPS-like planner in PROLOG. (Lin 2001)
[b] Heuristic Search Planner (2.0) (Bonet & Geffner 2001)
[c] "A little of this, and A little of that" (Nguyen, Kambhampati & Nigenda 2002)
[d] Greedy Regression Table (Refanidis & Vlahavas 2001)
[e] Intelligent Model checking and Planning System (Edelkamp & Helmert 2000)

Chapter 4

The Belief-Desire-Intention Model of Agency

In this chapter the theoretical model of agency based on the Theory of Practical Reasoning by Bratman (1987) will be presented. Various interpretations of the BDI model in the computer science will be used to illustrate it further. The chapter continues with special respect to the Jadex system as the primary target of this diploma thesis and with BDI systems that already integrated a planner in their architecture.

4.1 The BDI Model

The philosophical BDI model for a general type of an agent was developed by Bratman (1987).His work concerned with practical reasoning and the incapability of former belief-desire models to explain human notion of building plans and sticking to them. On the basis of his observations he concluded that plans are formalized intentions, which aid humans in their reasoning about the future by constraining the choice of their options. This conduct spares humans from inefficiently reconsidering all possible alternatives any time the world changes and it is opposed, this way, to decision theoretic models of rationality.

Whereas desires and beliefs are able to provide a reason for agent actions, they are not enough to explain the stability and efficiency of human reasoning. Bratman claims that both pro-attitudes including desires and intentions influence our conduct, but only intentions are responsible for its control. Intentions are not reducible to beliefs and desires, and must be understood as distinctive states of mind. Bratman points out two aspects of intention, the first being of volitional nature. It says that intentions are more than desires and provide more than a motivation to do something. Actually, intentions provide support for future actions and control their conduct. Contrary to agent desires, intentions should not contradict each other or the agent would be accused of irrationality.

The second is reasoning-oriented. It says that former intentions control the commitment to future intentions and restrict the amount of reasoning. This aspect cannot be reflected by a belief. It provides support for the expectation that an

agent will do something, but it does not entail the agent to fully believe it will succeed nor that it will fail. Bratman allows an agent to have intentions that stay in conflict with its beliefs.

In the view of Bratman, plans are not mere abstract parametrized scripts, but a form of mental states. As he says: "Plans [..] are mental states involving an appropriate sort of commitment to action" (Bratman 1987). The aspect of commitment makes plans more complex and structured intentions, build up from simpler smaller ones. This intentional structure is necessary incomplete as the agent is not required to commit to all options in advance. It is also hierarchical.

The theory of practical reasoning embedding Bratman's Claim found rather less implementations. One of such abstract true BDI system is the Intelligent Resource-bounded Machine Architecture described later. The main impact of BDI was on theoretical models of agency and rationality. Cohen & Levesque (1990) extended it to the domain of artificial agents and provided a formal model in modal logic facilitating this way its computational realization. There were other adaptations of this theory, mainly towards existing systems.

Georgeff & Lansky (1987) explained their Procedural Reasoning System in terms of the BDI theory and reduced the concepts to computational means[1]. "Beliefs are just some way of representing the state of the world ...". The world could be represented in this view using entities as simple as variables, relational databases or symbolic propositions.

But the notion of belief bears more than mere representation of information. Beliefs are distinct from objective truths as they belong to an agent. This subjective rationality of beliefs seeks the explanation of their emergence and dismissal. A fine representation of this structure can be given by forward-chaining networks with the prominent example in the SOAR system. The degree of belief, its importance to the agent and restricted inertia in the face of bounded resources must not leave the understanding of this concept. It is fairly easy to model the degree using multivalued logics (true, false, unknown), or use fuzzy logic to give it a sense of a mental state. Still, there is a problem in the view of a belief as a variable. Agents provided with an enormous number of facts cannot refer to each with a name and not every belief is prone to reification. Even the database approach reduces beliefs to the level of information. It simply requires the agent to know the structure and form of its beliefs in advance.

Desires have been reduced to a more concrete notion of goals represented by a variable or a symbolic proposition. Goal-oriented programming techniques are confronted with task-oriented ones. Conventional programs are task-oriented. They execute, what has been set at the time of program compilation. Goals allow agents not only to have the knowledge what to do, but also to include the reason for the execution of certain tasks (Georgeff et al. 1998). Further, goal-oriented behavior allows for more flexible failure recovery strategies in the light of unforeseen occurrences conflicting with current execution.

The need of goal orientation led in conventional programs to the emergence

[1] In a panel contribution on BDI (Georgeff, Pell, Pollack, Tambe & Wooldrige 1998).

of some programming patterns. This may be seen on the basis of the Chain Of Responsibility Pattern[2], Communication Filter Chains, and at last, event based processing commonly found in object-oriented programs. All of them include an object, message or event that must be processed by procedures not specified in advance. The assignment of it to the procedure is done at runtime. After a procedure has handled the object, event or message, it is marked as processed, changed or handed over to the next procedure. This has a strong notion of goal-oriented programming, as the object or event provides the reason for the execution and there are many procedures capable of handling it that need not succeed at their task.

Contrary to the meaning attributed by Bratman to plans, Georgeff sees plans as procedures or a "... generic, parametrized [...] special kind of Belief ..." that is carefully separated from other beliefs and stored for its use in the future. In his view, a plan becomes an intention if the agent commits to its execution. Intentions may be realized simply as "... a set of executing threads ..." embedded into the agent process. This does not depart from the understanding of intentions being partial and hierarchical as illustrated by the PRS system.

4.2 BDI Systems

There is a wide discrepancy on the question – what constitutes a BDI system? It is certainly the case, that due to the popularity of this topic and the metaphorical migration between the theory of human practical reasoning towards a computationally founded theory of agency, many different interpretations of these concepts emerged. In order to clarify the field a wide view of BDI systems by Martha Pollack will be adopted here. She divides BDI systems in three categories, depending on the model of agency they employ[3]:

- The Intelligent Resource-bounded Machine Architecture and other models adhering to Bratman's Claim in respect to practical reasoning;
- Architectures owning to the Procedural Reasoning System that is best explained using BDI terms;
- Models including beliefs, desires and intentions implemented on the basis of common understanding ("folk-psychology").

The following part of this chapter will present a survey of systems from all of these categories. It will start with typical BDI systems and continue with selected ones implementing AI planning techniques.

4.2.1 IRMA

The Intelligent Resource-bounded Machine Architecture has been sketched by Bratman, Israel & Pollack (1988). It presents a system implementing the BDI model and puts emphasis on the two processes responsible for means-end reasoning and

[2]Gamma, Helm, Johnson & Vlissides (1995)
[3]In a panel contribution on BDI (Georgeff et al. 1998)

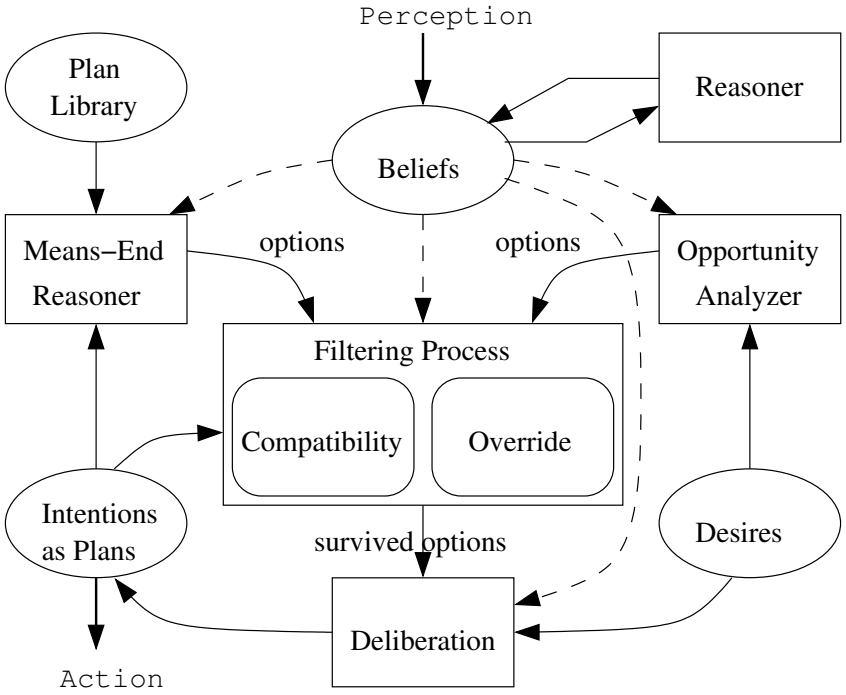

Figure 4.1: Intelligent Resource-Bounded Machine Architecture (Based on Bratman et al. (1988, fig. 1)).

weighting of solution alternatives. There is an assumption taken here about the deliberative process, responsible for choosing among options that may become agent intentions. This process is thought to be very time consuming, so IRMA includes a filter taking new options and proving their compatibility with existing agent intentions. This filter may be seen as the first stage of the deliberation process.

Figure 4.1 illustrates the architecture. An agent conforming with IRMA has beliefs, which are used by other components to guide their work. The beliefs are feed by perception and updated by a reasoner. The opportunity analyzer acts like a reactive component. It is influenced by desires and analyzes the state of the world as rendered by agent's current beliefs. Any time the analyzer finds a new promising option that would allow the agent to come closer to the fulfillment of its desires, it hands it over to the compatibility filter.

The filter, by itself, includes an override mechanism that allows important options to pass by and become available for deliberation. This override mechanism must be fine-tuned to the needs of the agent's application domain. Whenever such incompatible option passes the filter, the deliberation process must reconsider existing intentions and weight the costs and benefits of old intentions versus the new option. The purpose of this process is to assure that intentions structured into plans have no inherent conflicts dooming the plans to be irrational and provoke failures.

The last component of this architecture to be mentioned is the means-end reasoner. Its function is to work with the set of intentions, analyze their structure and propose options that would complete partial plans. The result of means-end reasoning is handed over to the compatibility filter and undergoes the same filtering

and deliberation process that was applied to options proposed by the opportunity analyzer.

4.2.2 PRS

Procedural Reasoning System is one of the best-known agent architectures mainly due to its pioneering role as a first practically applicable BDI-based system(Georgeff & Lansky 1986). The name of PRS derives from the term *Procedural Knowledge* defining agent abilities to act by invoking stored or learned procedures, as opposed to Declarative Knowledge describing facts about the world. PRS was devised to build systems for real-time applications that are non-stop situated in their environment and must exhibit intelligent behavior and use procedural knowledge as specified by domain experts.

Principally, it is a type of a hierarchical planner with well defined methods called here *plans*. It is bounded in its search as it works online and must *commit* to a plan as soon as possible. This type of planning decisions can be seen as *eager commitment* and is opposed to the term *least commitment strategy* conceived with non-linear planners. It is reactive as it does not anticipate future states in its basic algorithm and the deliberation must be done by specially provided meta-level procedures (meta-ACTs described later).

The architecture consist of a database where facts about the real world are represented by first order logic formulas constituting agent beliefs. The desires of an PRS agent do not represent classical declarative goals, but rather specify directly desired system behaviors and thus can be called *to-do* goals. This notion includes achievement, maintenance and query goals stated in form of temporal conditions over the future states of the agent and environment.

Like in most BDI systems, plans of a PRS agent are stored in a plan library. They were originally called *Knowledge Areas* (KA) and have been extended and renamed to ACTs – a common representation formalism for procedural knowledge used in the Cypress architecture described later. In fact the input ACT representation is being directly translated to the KA representation so the old PRS structure can use them internally. Knowledge areas are associated with an invocation condition that may be triggered by *to-do* goals, or by changes to the facts in the beliefbase. *To-do* allow for a sort of goal regression planning and hierarchical abstraction to be utilized by knowledge areas. Conditions on the beliefs resemble a reactive rule based agent architecture. By default, to keep the system reactive, no deduction or reasoning is used to activate the KA areas. The to-do goals are matched against an invocation conditions using unification. The other part of a knowledge area is its body, a plan schema, consisting of partially ordered sub-goals for the system. After being activated the KA's are stored on a to-do agenda of the PRS system and represent intentions of the agent. The agenda itself is partially ordered – owning to the partial order representation of KAs – forming an *intention graph* including suspended, delayed or deactivated intentions. Before an activated knowledge area can be chosen for execution, it must wait for preceding ones to be executed or dropped. Meta-level ACT's can manipulate this agenda by removing

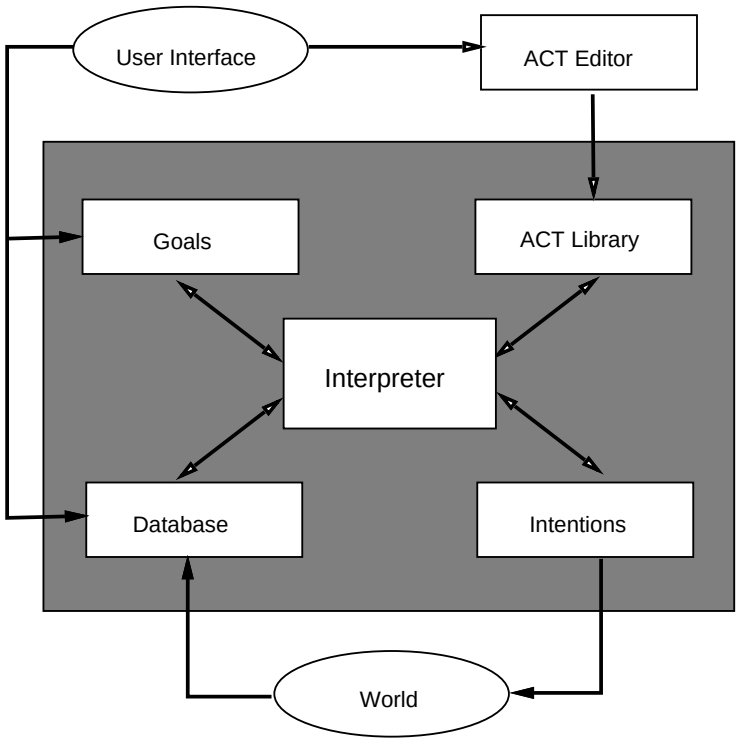

Figure 4.2: The architecture of a PRS agent (Myers 1997, fig. 1.1).

or adding knowledge areas and changing the order.

Figure 4.2 depicts a simplified view on the architecture of a PRS agent with components mentioned above. Additionally, it depicts the interface to the user controlling the agent and the domain of agent application. The PRS interpreter plays a central role as it communicates with, controls and manipulates the other modules and is responsible for changing beliefs, running KAs and executing actions in the environment.

The Procedural Reasoning System was successful in application domains including space shuttle diagnosis[4], network management, control of mobile robots[5] and joint military operations. The system has inspired many of the following agent architectures, especially architectures based on the BDI model of agency. Systems that claim direct descent are dMars[6], and Jack, which will be described later.

4.2.3 Jack

Jack stands for JACK Intelligent Agents™ by Agent Oriented Software (AOS) and is a lightweight platform and framework for agent development. It provides in form of plugins different reasoning models. Among them the *BDI* and *SimpleTeam*

[4]As Reaction Control System for the NASA space shuttle (Georgeff & Ingrand 1990).

[5]PRS-Lite controlled the Flakey autonomous robot at SRI International (Georgeff & Lansky 1987).

[6]distributed Multi-Agent Reasoning System – a C++ implementation of PRS (d'Inverno, Kinny, Luck & Wooldridge 1997).

reasoning models are present. The former is responsible for modeling agents towards reactive and goal-oriented behavior, the latter allows models of agents acting in teams. The pragmatic engineering concerns had brought within the JACK framework a new concept to agent-oriented programming. *Capabilities* term the new approach to modularize agents and facilitate software reuse (Busetta, Howden, Rönnquist & Hodgson 2000).

The framework is thought to be easily extensible and integrable into large legacy systems. In respect to these concerns, all agents can be viewed as a JAVA™ object and accessed by method calls from within an application. On the other hand, the communication paradigm of message passing – as used generally by agents – is not bound to a particular mechanism or language. Indeed, AOS claims that their agents – once devised to communicate with CORBA, HLA, DIS, PVM[7] or their home made fast message exchange infrastructure – are capable to use KQML and FIPA ACL protocols as well (Busetta et al. 1999).

In order to facilitate agent-oriented programming JACK does to JAVA™ what C++ did to C. The language is expanded by the following conceptual entities:

- *Events* are the cause for agent actions and reactive behavior.
- *Plans* are compiled procedures used to process events.
- *Belief-Sets* have been devised to come up for the shortcomings of JAVA™ in respect to knowledge representation abilities. They comprise a simple relational database approach to information storage and retrieval calling it *belief* in order to commit to the agent terminology.
- *Views* are defined over belief-sets and common object models and are used to retrieve information.
- *Capabilities* define reusable software components for agents.
- *Agents* itself include all of the above and represent a single execution entity in the system.

A precompiler uses files specified by the listed entities and produces JAVA™ output that is compiled into classes ready to be run on a JACK platform.

In the design of an agent the central role is played by its plans. Every plan handles one event – being a goal in the BDI terminology – and includes *relevance* and *context* functions. The former checks if the plan is relevant for a particular event, the latter evaluates the current situation and tests the preconditions. The body of a plan is made of JAVA™ boolean statements connected with an `and` operator. If one of the statements fails, the chain fails and the plan is recognized as failure. Meta-level reasoning is realized by *prominence* – the order listed in the agent file, or by *precedence* – the priority given to the plan by a programmer. If there are plans

[7] CORBA® – stands for Common Object Request Broker Architecture, HLA – High Level Architecture (IEEE 1516), DIS – Distributed Interactive Simulation (IEEE 1278), PVM – Parallel Virtual Machine.

with the same precedence, the case can be handled by a meta-level plan invoked by a *PlanChoice* event (AOS 2004).

Agent Oriented Software Group affirms JACK a wide basis of applications in military, industry and research. This includes applications in following domains: telecommunications, manufacturing, finance, air traffic management, aerospace, e-commerce, customer management, defense simulation and decision support, government services, education, product support. They claim their system to be excellent at modeling humans and interacting with them.

4.3 JADEX

JADEX (Pokahr et al. 2003) emerged from the MEDPAGE project – an agent-oriented application for treatment scheduling in hospitals. It was used for research, teaching and for specific purposes in different domains including agent-based simulation, multi-agent scheduling of business processes and mobile applications. There are demonstration examples including blocks-world, puzzles and other interesting agent domains. The mobile applications included a personal task planner and an intelligent travel assistant. JADEX was proved to be inter-operable with other agent platforms utilizing Agent Communication Language and FIPA[8] standards. In a hunter-prey scenario it was demonstrated to interact with agents on the CAPA platform (Duvigneau, Moldt & Rölke 2003).

The project team started with the JADE (Bellifemine, Poggi & Rimassa 1999) agent platform, one of the wide known agent frameworks providing FIPA compliant communication services and a rudimentary support for active objects. It became obvious that more elaborate models of agency where needed than the simple task model provided.

The extension to JADE provides a full blown model of agency with explicit goal representation, support for sets of beliefs, named beliefs and object-oriented representation of agent plans. The active attitudes of an agent are represented by events, which are created by incoming messages, thrown by executing plans and emerge from conditions posed over agent beliefs. They play a central role in controlling the work of an agent and glue all parts of the system together without forcing a tight coupling of the components.

Goals. Similar to events, goals are the motivational attitudes of an agent. They may be dispatched from plans, activated by create conditions. Goals give rise to goal events, handled by plans. Drop conditions describe, when the agent should stop pursuing a goal. Context conditions, specified over the beliefs are responsible for suspending a goal in certain situations.

There are four kinds of goals that may be given to the agent, as identified in the JADEX system. The *achieve* goals carry a target condition, which should be fulfilled if the agent succeeds on that goal. The *query* goals are similar to the achieve ones. Their purpose is to collect information. The *maintain* goal is the most complex

[8]Foundation for Intelligent Physical Agents.

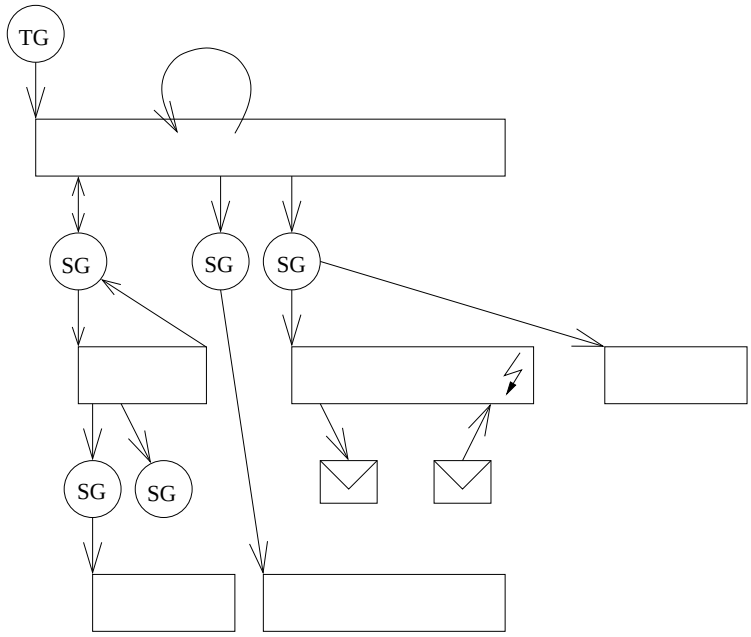

Figure 4.3: JADEX hierarchical plan decomposition. The plans are represented as a horizontal oblong and goals as a circle (TG – top goal, SG – sub-goal). The arrows show causal dependency among goals and plans, and illustrate the flow of control.

type. It is activated, whenever its *maintain condition* is violated, but stays idle after its target condition has been achieved. The last, but not least is the *perform* goal type. It simply motivates the agent to execute one or more plans (Braubach, Pokahr, Moldt & Lamersdorf 2004).

Goal deliberation. Contrary to former BDI systems, the goals, represented in JADEX explicitly, allow for a way to deliberate on them. Using constraints on goals, it can be chosen, which ones stay in the active state and which will be suspended. This, in other turn, elevates goals to the level of Bratman's desires and justifies the existence of conflicting goals in the system. This is unlike the Procedural Reasoning System and descendants where goals approximated desires, but were actually contained directly in the intentional structure of an agent. The latter required the designer to assure that no conflicting goals will be pursued.

Means-end reasoning. The approach to means-end reasoning is very similar to the one found in the PRS. JADEX uses eager commitment and plans with lazy decomposition. It is a sort of hierarchical task networks planning (cf. fig. 4.3), but without the look into future that would anticipate forthcoming events and the interplay of intentions and executed actions. This approach has the advantage of creating plans on the basis of full information that may be available to the agent.

The plans act as quite flexible decompositions and may include highly aligned domain knowledge, guiding agent intentions into right direction. The full power of the JAVA™ programming language may produce task networks that dynamically

adapt to the situation at hand and it gives the programmer a far more superior control over execution than the direct acyclic graph representation commonly used in the description of task decompositions.

As seen in Figure 4.3 the plans and sub-goals form the intentional structure of JADEX agents. The structure is decomposed in a hierarchy of sub-goals by the plans (procedures). As said above this plans may include all sorts of control statements and knowledge. As the plans are decomposed at runtime, they may directly influence the world model of the agent as stored in beliefs and they may take direct action towards the environment. This may include presenting new information to the user via graphical interface or communicating with other agents in order to influence them or query for information. Both goals and plans are tightly bound to their execution context by a set of conditions that control various aspects of the intentional structure. Goals may be dropped and plans aborted. Both may end with success or failure. The dynamic nature of this structure allows goals to be pursued by many plans in sequence or concurrently. The plans may dispatch many new sub-goals and wait for them or let them be fulfilled or fail unattended.

The disadvantage of this reactive reasoning model based on hierarchical decomposition comes from backtracking on plan failure and lack of anticipation capabilities. This shortcoming, inherited from the PRS system, is mainly due to the inability to project the world model towards the future and evaluate its various aspects over the time in advance. On the other hand, problem solving in hierarchies of goals reassembles the way humans usually try to cope with their own assignments.

4.3.1 Programming Model

To aid flexible and reusable software design JADEX agents are specified using an Agent Description File (ADF) and a number of JAVA™ classes determining the dynamic properties. This agent modeling approach is further enhanced with additional levels of abstraction. The developer may specify whole agent societies using deployment specifications similar to the ADF. These society files describe, which agents are necessary for the application and what are their interdependencies (Braubach, Pokahr, Krempels & Lamersdorf 2004).

ADF. The Agent Description File specifies the type of an agent. It mentions all elements of an agent explicitly, including *beliefs*, *goals*, *plans* and *events*. All of the elements are strongly typed to aid a clean software engineering approach. The file itself is written in an XML-based language defined by a schema. This bears the nice feature that a wide range of XML development tools may be used to write ADFs and check their syntax. The dynamic part of ADF is specified using a subset of JAVA™ language. This subset contains all right hand assignment expressions, plus some shortcuts to access array or vector data by index and bean properties by their name. Additionally, the user may resort to the Object Query Language (OQL) to access all sorts of beliefs and data in a systematic way.

Besides the basic elements, the programmer may specify other elements. This includes services and agent descriptions, ontologies and languages – all concerned

with the communication aspect of the underlying FIPA compliant JADE framework. For configuration purposes the ADF allows to specify agent properties. Import statements in the header allow for a concise description of JAVA™ expressions and OQL queries may be specified, so the agent loader will compile them in advance.

Capabilities. On the other side of abstraction scale, researches identified the need of modular design for agents. The concept of capabilities has been delineated and first described by Busetta et al. (2000). Capabilities in JADEX may be specified using XML in the same description file format like ADF. They include a clean interface facilitating the ability to compose the capabilities into agents in a hierarchy of dependencies (Braubach, Pokahr & Lamersdorf 2004).

Tools. The project is not only responsible for the core BDI reasoning system, but provides a number of tools to aid design, implementation and debug phases in the agent development process. Alone the underling JADE framework provides tools like the communication monitor - Sniffer, the Dummy Agent and the Remote Monitoring Agent. The JADEX projects extends this palette with BDI Viewer, Jadex Introspector, Logger Agent, Ontology plug-in for the Protégé Ontology editor and the Jadex BDI Tracer. JADEXDOC may be used to produce JAVA™ API-like documentation describing all elements constituting an agent as stated in the ADF.

4.3.2 Operational Model

After the Agent Description File has been loaded, the agent is represented by model elements describing the static properties. From this templates runtime elements are generated. The relation between model and runtime elements may be seen as one between a concept and an instance of it. For each concept there may be plenty of elements created at runtime. This aspects of JADEX builds directly on object-oriented design techniques.

Messages. Figure 4.4 illustrates that JADEX is a communication-oriented architecture. The messages coming from left in the figure are first assigned to a running conversation, provided they carry a conversation id. In the other case the *Message Receiver Behavior* matches them against event templates stored in the different capabilities of an agent. Thereafter a message event is generated and dispatched to the corresponding agent capability.

The reaction and deliberation modules are responsible for finding plans (from plan library or running) and to handing over the event to them. The deliberation module will choose from adopted goals and activate or suspend them accordingly to the deliberation policy. The reaction module evaluates the state of an agent, generates goals, processes events or instantiates plans.

The running plans are stored in a *ready list* from where they are selected for execution by a scheduler. As shown, plans may influence the agent subsystem by dispatching sub-goals or generating internal events. Plans are responsible for

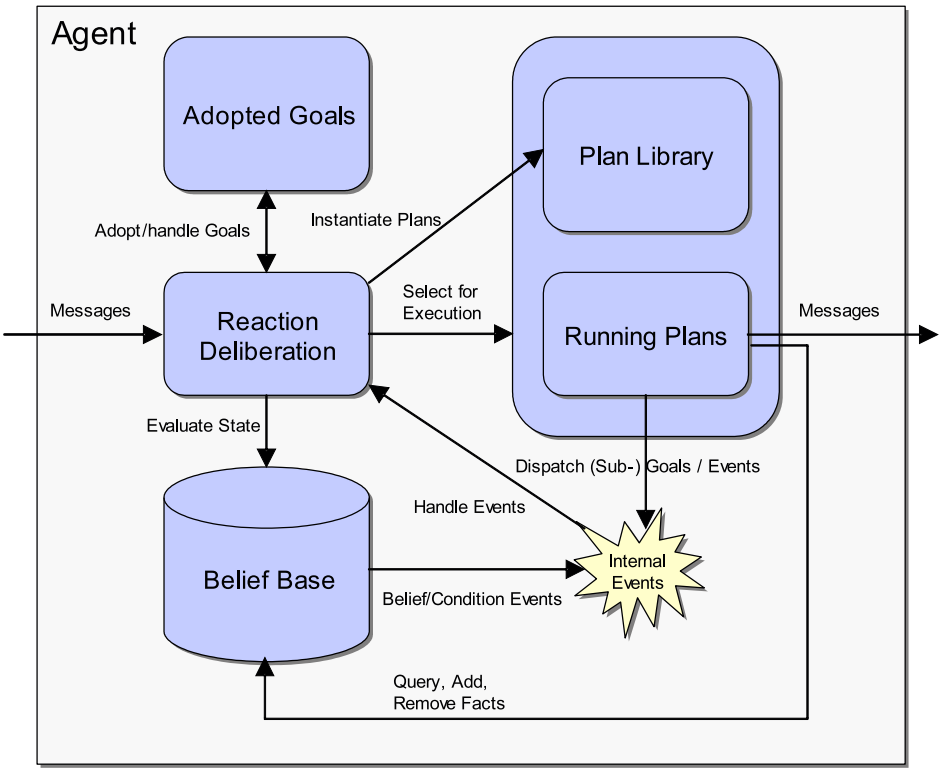

Figure 4.4: JADEX architecture (Braubach, Pokahr & Lamersdorf 2004, fig. 1).

reaction, domain specific reasoning, communication and access to legacy systems using JAVA™ API.

The *belief base* plays an active role in the JADEX architecture. It is responsible for generating events concerned with beliefs and conditions. Running plans may query the belief base using OQL. They may add, remove and modify facts stored there.

4.4 BDI Systems with a Planner

Several BDI systems have been extended or designed with the notion of artificial intelligence planning. This section will list few important examples.

4.4.1 INTERRRAP

The INTERRRAP architecture is a successor of the *Reactive Action Package* (Firby 1989) system that was used to control large scale real-time applications. Contrary to the latter one, INTERRRAP was designed in the notion of the BDI terminology and extends the RAP system by introducing a layered design. There are three layers employed. The first one called *Behavior Based Layer* (BBL) is responsible for reactive behavior to situations in the environment that require immediate action from the agent. The second layer is named *Local Planning Layer* and is responsible for agent reasoning about future actions that should be taken in response to goals

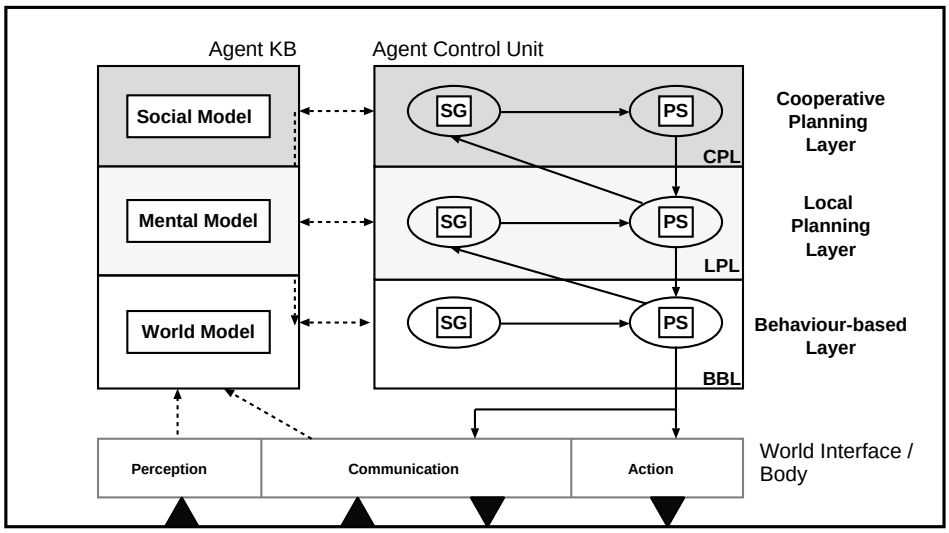

Figure 4.5: The architecture of an INTERRRAP agent. Dotted lines represent information flow. Full lines demonstrate the flow of control (Fischer et al. 1994, fig. 1).

posed by the environment and the BBL. The last layer concerns agent attitudes and intentions in respect to other agents in the environment and is called *Cooperative Planning Layer*.

In Figure 4.5 the layered approach is exemplified. At the bottom there is a layer dealing with the surrounding world and communication with other agents. All interaction starts at this layer in form of events referring to messages or fact changes. The events are propagated layer for layer in the upward direction through the knowledge base. At every layer they are matched against first order formulas and produce *situations*, which principally describe a part of the environment or agent internal state. The situations are mapped again to goals building a *situation+goal* tuple $< S, \gamma >$. The process of situation recognition and goal formulation is done by a unit annotated with SG in the figure.

The situation+goal tuples are forwarded to the planning, scheduling and execution component (PS) at each layer. Each PS is equipped with a competence function to decide if it can handle the situation. Provided, the layer is competent of this situation, a course of action will be devised by this component and the layer will commit to this actions. It will forward them as intentions to a layer below, so they can be integrated into the intention structure. In the other case, the $< S, \gamma >$ pair must be handed over to the layer above with the hope it will be processed there and produce results. Intention structure is generally represented as a partially ordered set of actions that are executed by the BBL.

The Behavior Based Layer includes hardwired links between situation-goal descriptions and actions to be performed, called *Patterns of Behavior*. Processes at this level are in accordance with the Markov property requiring that any action at the time point t_i depends only on the state of the world at t_{i-1}. The layer may be implemented using the RETE algorithm, a fast forward chaining approach that

allows the agent to remain as reactive as possible.

The Local Planning Layer includes a single domain dependent planner to solve the goals. This may comprise a plan library approach with many plans capable of solving given goals and a deliberation algorithm choosing among the plans. Other scenarios include a hierarchical decomposition planner. The Cooperative Planning Layer is devised to cope with issues concerning multi-agent systems and acting in multi-agent environments and is an answer to the shortcomings of the BDI model that fails to cover this topics in its theory.

The *Knowledge Base* of the agent is layered as well. This lowest level accessible only by the BBL contains a model of the environment including static declarative propositions and dynamic descriptions of physical processes. The *mental model* includes agent beliefs about itself. Especially the particular abilities and resources available to the agent are represented here. This knowledge level as well as the world model can be accessed by the Local Planning Level. The *social model* is the last level of agent knowledge. Facts about other agents are represented here. The goals, intentions and capabilities of other agents are stored here together with *joint plans* the agents might have agreed thereupon. The Cooperative Planning Level may access the social model and all underlying levels of knowledge representation.

The function of InterRRaP could be demonstrated in an automated loading dock domain. This application consisted of simulated and real miniature robots with the task of loading or unloading a truck. The main problem of this domain was the conflict avoidance among singular robots each of them controlled by a single InterRRaP agent. The resolution mechanism situated in the CPL showed particular feasibility for this domain (Fischer et al. 1994).

4.4.2 Cypress

The Cypress agent architecture and integrated planning environment is named after two main components Sipe-2 + PRS. It was the first system claiming to integrate (re-)planning, control and uncertainty reasoning capabilities (Wilkins, Myers & Wesley 1994). This is achieved by incorporating mature technologies for planning, reacting and reasoning into the system and gluing them together with an interlingua, a form of representation, called *The ACT Formalism* (Myers & Wilkins 1997). The system has been applied to many problems including military operations and fault-diagnostic.

The components are depicted in Figure 4.6. The Sipe-2 propositional planning system is responsible for generating plans in form of ACTs that are transferred to the PRS module for execution. PRS is responsible for acting and reacting in the agent environments. It does so by executing plans generated by the planner or prepared by the domain writer, who is aided by an ACT-Editor in the process of domain modeling. Both systems are capable of handling the ACT formalism as their input language. Before use, Sipe-2 transforms ACTs into its internal operator representation and PRS transforms them into knowledge areas.

PRS is responsible for monitoring the environment and responding to the changes. If in the course of execution some sort of unexpected events or impasses

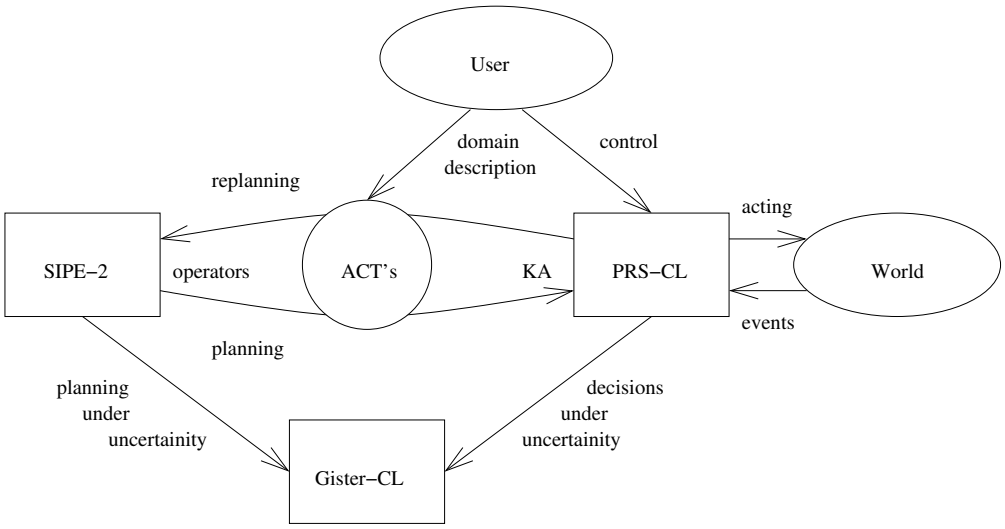

Figure 4.6: The architecture of a CYPRESS agent.

arise preventing the system from proceeding with current plans, the execution of such problematic plan is halted and the problem is turned over to the SIPE-2 planner for replanning. While new plans are generated PRS continues with execution of plans unaffected by the event or impasse. When SIPE-2 has finished its replanning process it must compile its new plans into ACTs and transfer them back to PRS where the ACTs are merged with plans on the current intention stack.

The task of planning and executing is divided by an interface defined on a specified abstraction level. Goals, as described by ACTs, are placed on different levels of abstraction, demonstrated by their appearance on different levels in plans. There is a fixed number of the abstraction levels (e.g. nine levels for a military operation scenario). Goals on higher level are handled by SIPE-2, which generates plans composed of goals only from levels below the interface. PRS is devised to handle only lower ACTs and goals.

CYPRESS has been extended with the ability to plan and react under uncertainty. Both main modules, PRS and SIPE-2, resort to the GRISTER-CL module that utilizes the Dempster-Shafer[9] and Bayesian probabilistic models in order to provide a suite for evidential reasoning. It is able to draw conclusions from multiple sources on the basis of evidential information. Possibilistic models, including propositional and fuzzy logic, are also integrated into the system through this module.

4.4.3 RETSINA

Reusable Environment for Task-Structured Intelligent Networked Agents is a multi-agent infrastructure developed at CMU in the Software Agents Lab. The system consist of agents – concentrated on message processing – composed into an extended model-view-controller (MVC) pattern.

RETSINA multi-agent systems include:

[9] *A Mathematical Theory of Evidence.* (Shafer 1976)

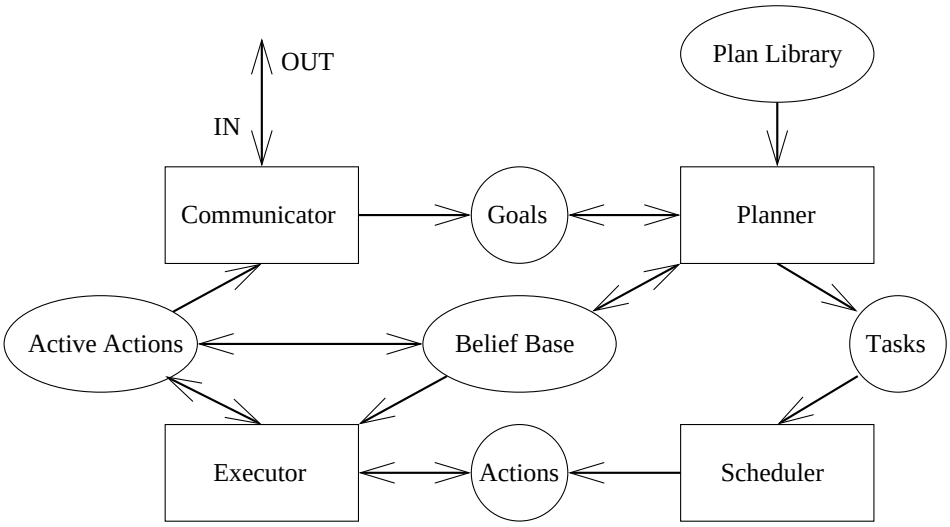

Figure 4.7: The architecture of a RETSINA agent.

- *interface agents* to interact with users,
- *task agents* providing services like information processing, planning and problem solving,
- *information agents* storing and retrieving information relevant for the task agents from data bases and legacy systems,
- *middle agents* provide the glue between the other types of agents.

The last type of agents represent the dynamic extension to the MVC pattern and is a result of system's agent-oriented aspects (Sycara, Paolucci, Velsen & Giampapa 2003).

A single agent consists of (cf. fig. 4.7) a communicator receiving messages from other agents, a hierarchical planner using a plan library, a scheduler and an execution and monitoring module. The communicator introduces *objectives* into the system on the basis of received communications. It is also responsible for sending messages if instructed so by the actions. The planner takes the objectives, kept in a priority queue, as its goals and uses an HTN approach in order to find a solution by decomposing abstract tasks into primitive ones. The output from the planner is posted onto a task agenda, which is used by a scheduler to compose them into a timed action sequence. The heading actions become activated by placing them into an *active* pool where they are executed and monitored by the last module. The effects of these actions may include manipulating the belief base, changes to the schedule, tasks and objectives as well as sending messages to other agents.

A planning domain for the planner is defined by a tuple $< A, C, R >$ where A are primitive tasks, C are complex tasks and R are reductions describing how to recursively decompose elements of C into elements of A. A planning problem is stated by $< B, O, T >$, with B constituting the set of beliefs, O describing the goals to be accounted for and T being a set of task structures representing partial plans.

The planner proceeds by matching an objective with task decompositions from R, contributing to the set T of initial plans. It then repeatedly chooses an element of T and removes flaws from within, producing new refinements that are added to T. If the plan chosen contains no flaws it is returned by the planner and the primitive tasks become actions presented to the scheduler.

Partial plans contain three kinds of *flaws*:

1. Task reduction flaws represent complex tasks, for which no reduction has been found yet.

2. Suspension flaws are introduced into a plan if a given reduction cannot be performed because a *provision*, required for a task from within the reduction, is not available yet.

3. Execution flaws represent causal dependencies between suspended reductions and active actions being monitored.

Both suspension and execution flaws are responsible for the capability of RETSINA agents to interleave planning and execution. Provisions, mentioned above, are data required by the planner to evaluate the effects of a task, its outcomes and constraints posed upon it. The planning may continue with portions of the plan independent of actions querying for the missing information. The RETSINA planner can be seen, this way, as a successful compromise between eager commitment reactive systems and least commitment offline planners.

The agent uses special cyclic actions called monitors to check the validity of plan constraints. If the plan under consideration is still a partial solution, it is removed from the set of valid plans. If a constraint of a scheduled action is violated, the corresponding plan is replaced with a new one. Active actions with violated constraints are permitted to finish their execution and return success or failure. The last case would consequently make the relevant plan invalid and stop its execution (Paolucci, Shehory, Sycara, Kalp & Pannu 2000).

The RETSINA system and the multi-agent infrastructure was applied to numerous domains. This includes portfolio management, content management, auctions, logistic, military operations and mobile communication. It was shown to be interoperable with other infrastructures on different types of communicating platforms (Sycara et al. 2003) including grid and p2p networks.

4.4.4 DECAF

The Distributed, Environment-Centered Agent Framework[10] has been devised to develop multi-agent systems written in JAVA™. The authors claim following capabilities provided by the agent architecture: ".. communication, planning, scheduling, execution monitoring, coordination, and eventually learning and self-diagnosis .." (Graham & Decker 2000). The perceive-plan-execute cycle and the planning component is very similar to the one situated in RETSINA agents.

[10] A framework from the University of Delaware.

The agents in DECAF are defined by a plan file specified using a visual tool called *PlanEditor*. The file contains task and action definitions referencing JAVA™ classes and their methods. Each action is defined as a method and each task is a class file containing many actions. The plan file does not specify any goals nor beliefs of the agent.

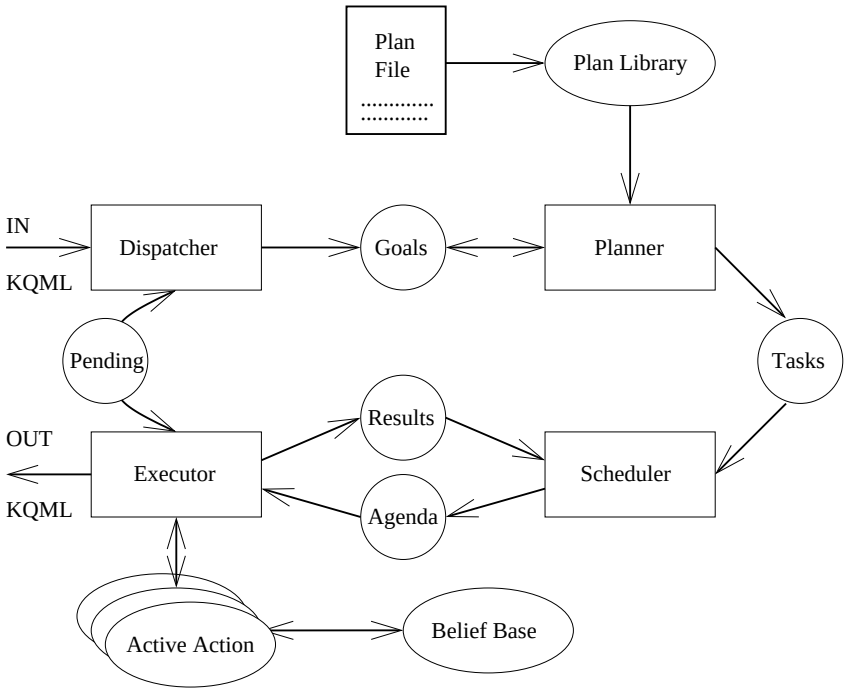

Figure 4.8: The architecture of a DECAF agent (based on Graham & Decker (2000, fig. 2)).

After being initialized, the agent possesses a number of task templates and a set of beliefs modeled by JAVA™ data structures. The *objectives* (goals) of the agent are posed by incoming KQML messages. As shown in Figure 4.8, the *Dispatcher* is responsible to process the messages, decide if they concern pending actions or otherwise create a new goal and put it into the queue of objectives.

The *Planner* takes the objectives out of the queue, matches them against task templates and instantiates the ones, for which there are enough provisions. Provisions are given by the KQML message and by the results of actions executed. Instantiated task templates go onto the task queue where they are taken over by the *Scheduler*. There is no global task structure, holding the agent's intentions as a composed task network that is processed by the planner. The work of the planner does not go beyond estimating preconditions, selecting task templates and instantiating them. This makes DECAF a reactive eager commitment system without inherent ability to dynamically anticipate future events.

The job of the *Scheduler* is to prune the tasks out of irrelevant actions and to determine, which of the relevant ones may be executed right now. If there are enough provisions for an action, it is placed onto the *Action Agenda*. Otherwise, the Scheduler waits until a result is returned by an executing action. The *Executor* calls

the methods denoted by actions on the agenda and returns their results back to the Scheduler. The actions access beliefs of an agent and may send KQML messages.

The DECAF architecture was devised to provide an operating system level for third party agent applications and as a research platform. It has been applied by following projects (Graham, Decker & Mersic 2003):

- Virtual Food Court – small economy simulation,
- Generalized Partial Global Planning – a centralized approach to planning and coordination in multi-agent systems,
- GeneAgent – an information retrieval system based on biological databases from over the internet.

4.4.5 PROPICE-PLAN

PROPICE-PLAN is an agent architecture based on a procedural reasoning framework called PROPICE. PROPICE – a descendant of PRS – is a commercially used product written in C programming language. PROPICE-PLAN extends the underlying reactive system with two capabilities. At first, it introduces a planning component capable of *plan synthesis*. The second extension is responsible for *anticipation planning*, a forward space search used to envision future situations and estimate best actions to do.

The central role in the extended system plays a common representation for *operational plans* (OP). All components use a textual description of plans similar to the one used by PRS extended by an additional field representing declaratively the *effects*. A simple procedure for moving a block would be written like below:

```
(defop |Move Block|
   :invocation (achieve (on $block $destination))
   :call    (Move-Block-S $block:Block $destination:Surface)
   :context ((test    (on $block $below))
             (achieve (clear $destination)
             (achieve (clear $block))
   :effects ((add (clear $below))
             (del (clear $destination))
   :body    ((call $block.move_to($destination)))
```

Invocation describes the main effect of an OP. There may be many OPs with the same invocation and the *body* part of a hierarchical OP includes invocations to other ones. *Context* includes filter expressions (denoted by `test`), preconditions (that may be achieved by another OP) and invariants stated over the course of a plan. The *call* field is used to identify an OP, to bind its variables and to instantiate it. *Effects* imitate the PDDL representation and include add, delete and conditional effects. This field is required by the planner. The *body* may include lists of OP invocations, conditional invocations, concurrent invocations, loops and calls to predefined procedures.

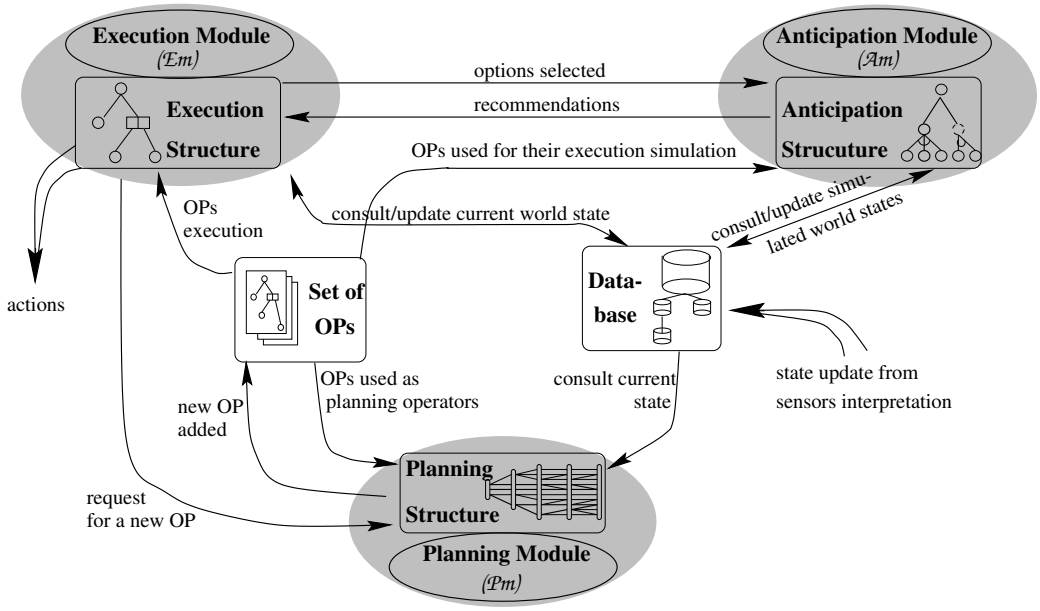

Figure 4.9: The interplay of execution, planning and anticipation modules in a PROPICE-PLAN application (Despouys & Ingrand 1999, fig. 1).

As shown in Figure 4.9 the legacy system consists of an execution module Em, a plan library and a data base. The choice of options – presented to Em by the plan library in response to an external event or a goal – is guided by the anticipation module AM. The latter simulates different executions of plans, estimates their effects and future requirements, prevents death ends and creates advices to be used for the former.

The different simulation traces are kept in a tree-like *anticipation structure*, with branches corresponding to different choices of plans and unknown results of sensing actions. Over the course of simulation optimality criteria can be applied and safety conditions preserved. The Am module uses the data base to create and store dynamically the evaluated future states and links them with the anticipation structure. On the other hand the Em is responsible for informing the Am, which choices have been taken and which structure branches may be pruned.

Whenever the Em faces a goal, for which there are no applicable plans in the specific situation, it must ask the planning module to produce a new plan out of present procedures. The planner used for the purpose is attached by an interface using the ADL action semantics. In particular PROPICE-PLAN delegates the plan synthesis task to the IPP planner described earlier. There are slight modifications to the algorithm in order to preserve domain dependent safety conditions while planning (Despouys & Ingrand 1999).

Obviously PROPICE-PLAN composes reactive procedural reasoning techniques with offline planning algorithms in a clean separate way, with plan library and OP notation defining the interface. The approach has been criticized, mainly due to the separation of planning from execution, and the anticipation module claimed to be an overhead (de Silva & Padgham 2004). But in the domain of application (furnace

control) it did not turn out as a handicap for PROPICE-PLAN agents, as there was plenty of time to plan and anticipate between the actions executed. On the other hand, the agent had to react timely in seconds to occurrences from the controlled machinery and could use well the advice prepared in advance by *Am* .

4.5 Summary

The Theory of Practical Reasoning by Bratman (1987) provides computer science with an application adequate model of agency. It is explained on the basis of the Intelligent Resource-bounded Machine Architecture. The BDI model offers reactivity and efficiency of reasoning that was not given by former deductive and deliberative architectures. The success of this model prevails up to this time in many new implementations. Especially the Procedural Reasoning System and its descendants use the BDI model and advance it further. Most recent systems include JACK™ agent development framework and the JADEX BDI reasoning engine.

The architectures INTERRRAP, CYPRESS, RETSINA and PROPICE-PLAN described above included a planner at different points in their agent operational model. They all had in common a declarative representation of goals, inherited from the PRS and as simple as `(test ?block clear)` or `(achieve ?a on ?b)`. This declarative structured representation has the advantage of being easily processed with standard means like unification.

INTERRRAP is a layered architecture where the local planning layer may include a symbolic planner, but no details have been given in literature as to the integration of such a planner, beside the situation-goal control cycle. CYPRESS follows another approach. It defines a common language that must be understood by the PRS and SIPE-2 system. It is a top-down approach where the planner controls the reactive system.

PROPICE-PLAN uses an another approach where PROPICE (a PRS-like system) asks a planner to create plans out of simple methods. For this purpose, the declarative representation of a method is extended by a single field describing its effects. PROPICE-PLAN stores created plans in the plan library.

RETSINA agent architecture has been devised with the notion of artificial planning as its fundamental concept. In fact, this design does not differentiate between the PRS-like execution mechanism and the planner posed as a central component. Both are one and the same. The representation used for plans and goals is the one of an artificial intelligence hierarchical planner.

DECAF system claims to be very similar to the RETSINA architecture. Unfortunately, the author failed to find any evidence of a symbolic planner in the literature and code of the DECAF framework. No results can be derived from this system for the further course of this thesis.

Chapter 5

Design

The choice of planning techniques and integration issues will be justified in this chapter. Some aspects of the implementation are used to substantiate the view. The chapter includes a resume on a partial order prototype that was created at the design stage and failed the purpose of this thesis project. On the other hand, it influenced further design and implementation and deserves a short presentation.

The questions regarding building a planner are: What it will plan for? How states and changes will be represented and how do the goals will look like? Section 5.5 presents answers based on ideas that fairly extend the representation applied by planning languages like PDDL and allow to state most of the planning concepts in JAVA™.

The design of fundamental things like *term representation* has been done at the very beginning of this project, as its structure was not known yet and it was unclear how to represent and process symbolic information in the JAVA™ language. Therefore, it was performed in a prototypic process.

The representation of JADEX terms, the unification algorithm and interpreter are kept as general as possible and may be used for different kinds of reasoners. This symbolic manipulation kernel has been divided in two packages:

- `jadex.planning.common` collects all common manipulation routines that are language independent.
- `jadex.planning.java` contains routines and an interpreter capable to handle the JAVA™ expression language.

The design of a search algorithm and a state space planner is built atop of this symbolic manipulation kernel. The last section describes the ideas and design choices taken in respect to the integration process of the planner and the JADEX reasoning engine.

5.1 Related Works

Most agents including a planner have been designed and implemented with the notion of planning in mind from the very beginning. Most architectures with strong

emphasis on artificial intelligence include a planning component as their central part. Such a design forces the system to use a representation of procedures, actions and beliefs required by the planner to be stated in a declarative way. This allows to access their description using standard artificial intelligence tools like unification. Such an example is the InterRRaP system (Fischer et al. 1994) described in Section 4.2. It has the notion of BDI and includes a local planning layer built upon an hierarchical planner. The representation of procedures and goals follows that of an HTN planner and has a declarative form aligned with planning.

Reactive systems, such as PRS, have been developed in response to the unacceptable lack of efficiency presented by planners at that time. It was more practical to write the plans by programmers and augment them with domain specific control knowledge. Nevertheless systems have been built that joined both paradigms. This proved especially useful in domains featuring enough time for planning, like in the example of Propice-Plan. It extends the PRS-like dMars system with a state based planner IPP. The composition was particularly successful, because both systems are implemented in the C programming language and the declarative PRS notation for goals and procedures needs only to be slightly extended in order to fit into IPP (Despouys & Ingrand 1999).

The Cypress system constitutes a marriage of a hierarchical planner Sipe-2 and the PRS system, both implemented in common lisp. Although both use fully declarative representations of goals, methods or procedures a new common language (called The ACT Formalism) had to be devised above both systems, gluing them together.

There are many planners implemented in high-level languages allowing for functional or declarative abstraction and easy definitions of meta interpreters. These languages already feature most tools needed to represent knowledge and reason about. Some of the recent planners have been implemented in imperative languages like C and most of the time showed high performance increases, mainly because they have been compiled to machine level and are not interpreted.

There are few planners implemented in Java™. One of them is the hierarchical planner JShop. It is a reimplementation of Shop, which in turn is based on Lisp. The input language is in form of HTNs notated in a declarative way and gathered in a domain file. There is no interface between the language processing part and the planner itself. JShop does not reflect the planning information in data structures, but works on the underlying textual representation in a symbolic way. Using JShop means writing all procedures in a Lisp-like manner and providing them as a domain file. A planner compilation technique has been developed for the Shop2 system that converts problem description from the domain file into Java™ domain specific planners. This approach promises to yield very efficient planners, yet it restricts the user again to state the domain in JShop specific language (Ilghami & Nau 2003).

There is a work comparing the aspects of planning in JShop and the BDI-like Jack™ system. de Silva & Padgham (2004) found a way to recode, presumably by hand, the blocks-world examples from JShop into Jack™ plans written in Java™ and compared the performance in time of both systems. It was concluded that Jack agents are faster and require less memory, even if JShop was forced to reassemble

the BDI execution mechanism with respect to eager commitment.

The $\mathcal{TAL}$PLANNER, a system based on the ideas of $\mathcal{TL}$PLAN[1], has been implemented in JAVA™. The input language has a declarative form and the planner requires control knowledge stated in $\mathcal{TAL}$ – a temporal logic language. This planner has been written around a simple search algorithm (Kvarnström & Magnusson 2003) and most of its internal processing actually concerns the temporal knowledge about the domain. The sources are not available for insight to a wider audience yet.

There is a rather theoretical work concerning the mapping of BDI internal mental states to a STRIPS-like notation and back (Meneguzzi, Zorzo & da Costa Móra 2004). This is done on an abstract BDI interpreter called X-BDI (Móra, Lopes, Viccari & Coelho 1999) implemented in PROLOG and augmented with the GRAPHPLAN algorithm written in C++. The representation of mental states of an X-BDI agent is fully declarative. The mapping is a structure transformation of beliefs, desires and intentions into a propositional notation that is used by the planner so the beliefs and actions must comply with the STRIPS domain representation.

5.2 First Approach

At the beginning the diploma project aimed at the development of a full symbolic planner, based on the partial order hierarchical representation. This proceeding was supported by modern textbooks on planning[2]. The idea was to use the planner and reason about BDI goals and plans and provide compound plans out of this basic components. In particular, all JADEX goals include the context and drop conditions and – what is more interesting – achieved goals may carry a target condition. Augmented with preconditions taken from plans referring to these goals, an abstract strips operator could be created as a tuple $< g, p >$ consisting of the goal and the referring plan, for each of two. This representation could be fed into the planner in order to create chains of these abstract operators.

Chaining of methods with help of preconditions and effect conditions may be seen in the PROPPLAN system. The developers of PROPPLAN had the advantage of provided propositional representation for goals and plans, given by the PRS representation of procedures. The goals in PRS have the form of a proposition as simple as `(achieve clear x)` and conditions are simple conjunctions of such propositions. The domain of furnace control provided the planner with enough time to create its plans. The planning component could use about eight hours for this purpose (Despouys & Ingrand 1999), so optimum planning speed was not a concern here.

Some problems arose by the application of this approach to JADEX architecture. First, almost no example of JADEX application domains used a precondition for the plans, so this approach would not scale backwards. Second, the expressions used in goal target conditions, have been far more complex than expected and a simple attribute variable state representation commonly used by planners would not suffice

[1] A winner of the International Planning Competition 2002.
[2] Russell & Norvig (2003).

to reason about it. The target conditions would have to be stored in a planning database – one for each planning step – in their original form, as structures prone to unification. Additionally, a meta interpreter had to be implemented to handle a lot of cases like

$$\texttt{battery} == 1.0 \implies \texttt{battery} <= 0.5$$

in a generic way. This proved to be difficult in language like JAVA™ and not efficient at all. The means and tools for this type of symbolic reasoning have been provided in the early stage of the diploma project and are partly used by the new design.

Third, many of the condition expressions heavily used calls to functions implemented in JAVA™ and compiled to JAVA™ byte code, which representation was not accessible to the planner and certainly not open for efficient reasoning with. Although, there is an implementation of a JAVA™ interpreter provided by the diploma project, it was surely not the concern here to provide a new JAVA™ virtual machine.

This negative result is also due to the complexity of partial order planning and due to the complexity of reasoning with full blown JAVA™ language like structures, which required the use of costly unification and reflection at many points of the planning process. After testing this first approach the author deleted most of the sources because of the poor performance. The project continued with investigation of more successful recent techniques in use.

5.3 Rationale

Due to the affinity with the BDI model of practical reasoning, PRS-like systems work with the *eager commitment* principle using ordered decomposition and replanning on failure. On the other hand HTN planners work using the *least commitment strategy* because of the partial order representation. The former ones work well in domains with retractable sequences of actions or where knowledge about best practice can be well encoded in plans by the agent designer. In domains where effects of actions cannot be easily retrieved or the agent must limit its try and error behavior, the best sequence of actions should be known in advance. The limited capability of foresight given to a BDI agent and the impossibility to provide all plans needed for the future at the point of creation compels the agent designer to revert to means-end reasoning techniques.

The planner implemented into the JADEX system has been guided by the recent development in the planning community. The idea was to take leading planners demonstrated at scientific planning competitions like the International Planning Competition and construct a planning kernel based on the ideas that allowed them for the accomplishments. This approach had particular advantages:

- The planners have been practically tested on standardized planning examples.
- It may be assumed that extraordinary performance indicates the right choice of planning algorithms.
- Only planners that are easily expandable and adaptable to new domains can

participate and face new challenges posed at all times to planner developers by successive competitions.

The argument for a knowledge controlled planner is quite intuitive one. The most successful problem solvers known up to date do not possess a universal capability to solve any problems from the very beginning, but must acquire lots of domain knowledge in a long process of learning. Again, taking under the scope static plans used in PRS-like systems are all very domain dependent and contain in their principle procedural knowledge, which is directly coupled with the domain of execution. It is assumed that providing control knowledge for the planning domain is only slightly more costly than stating it in the procedures or BDI plans.

Currently there are two successful approaches to planning that strongly build up on the domain knowledge. HTN planners include – beside the operator preconditions common in every planning representation – also a lot of knowledge stored in the *methods* guiding the decomposition of higher level actions. Forward search state space planners like $\mathcal{TL}$PLAN use domain knowledge in their control rules posed over the course of actions that may be aligned in the plans created.

The system implemented uses the second approach, as it has proved to be easier to implement. Additionally, the state space approach seams to be more intuitive for the type of imperative languages, because it works directly with the concept of state and concerns its manipulation in a way similar to the imperative programming paradigm. Also this choice derives from the fact that JADEX – as most other PRS-like systems – already includes a very simplified hierarchical planner. This thesis seeks to extend the system with a planner and not to reinvent the PRS algorithm at stake.

The approach taken in RETSINA felt out of the thesis scope, due to the same concern. In RETSINA a complete HTN planner has been taken as one of the central components from the very beginning and modified to provide more reactive behavior, without the loss of foresight and the capability to create dynamic plans in advance.

The solution presented here is quite the converse of the one in the CYPRESS system. CYPRESS has an upper planning layer handled by the SIPE-2 planner and a lower layer providing reactive and execution capabilities handled by the PRS system. From the view of a programmer, there must be knowledge provided in form of decompositions for the planner and for the PRS system. The plans created by SIPE-2 are simply viewed as a chain of top level goals in the PRS. The declarative ACT Formalism is a simple convenience wrapper for the programmer, so both underlying formalisms from SIPE-2 and PRS do not need to be considered and learned.

The approach here cannot take an advantage of common declarative representation for procedural and planning knowledge as most of this knowledge is compiled into JAVA™ byte code. To use a hierarchical planner like JSHOP above the reactive PRS reasoning algorithm would require the user to write method decompositions in LISP-like code, which conflicts with aims of this project. On the other hand, the planner developed here may be applied at any level in the goal execution hierar-

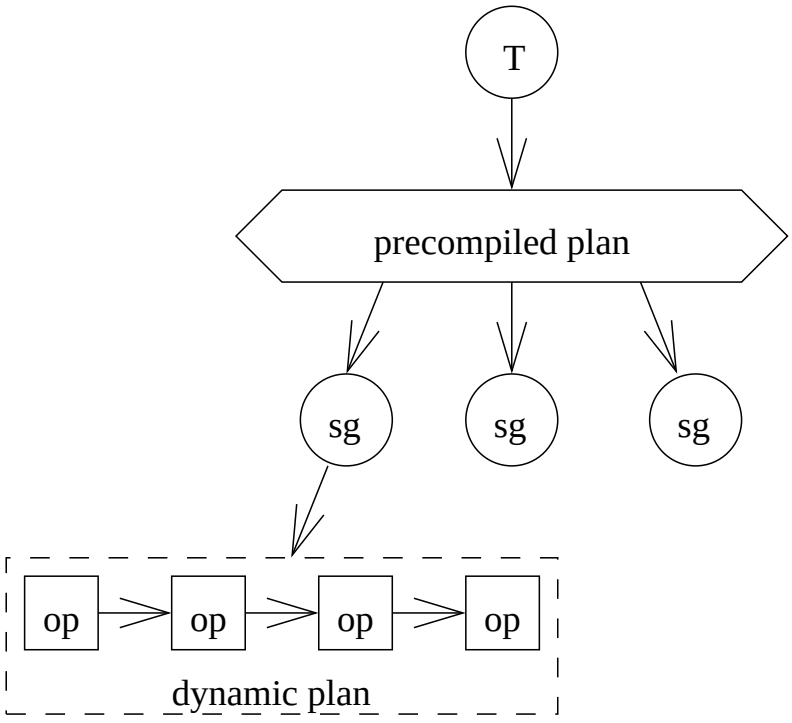

Figure 5.1: The dynamic plan created at runtime from planning operators (op) can be instantiated by a subgoal (sg).

chy. Most notably it may create plans that directly manipulate the beliefs at the lowest level of execution or it may create plans that spawn sub-goals and evoke a hierarchical execution process.

As described earlier, the method chosen here is based on a simple state space search augmented with domain knowledge. This approach is quite intuitive for a domain modeler operating with imperative languages. The states, to which operators are applied are always fully specified with respect to object attributes and variable bindings. Further, this method has been thought to function as a supplement to the hierarchical goal-oriented execution model presented in PRS-like systems. The high efficiency of this method allows to use the planner at the lowest operational level directly induced by sub-goals created from static plans and procedures. This approach is illustrated in Figure 5.1. The name dynamic plan derives from the fact that the operators do not form a precompiled sequence in advance, but are aligned at runtime to achieve the goal posed.

Contrary to the idea taken by PROPICE-PLAN where plans created by the planner are stored in the plan library, the approach here does not preserve dynamically created plans to use them in the future. This may be supported by the fact that the plans are crafted for a special situation and target and it is difficult to adapt them to new planning problems. The other argument is of pragmatic nature. If a problem and solutions to it are of general nature and the plans may be reused, the agent programmer is certainly more capable to write a static plan and store it in the plan library. With dynamic plans for infinite domains the plan library would be

over flooded by plans created for every possible planning problem, most of which would never be used again.

The planner implemented here provides planning, scheduling and simulation capabilities to the JADEX system without loss of performance and effectivity. It works supplementary to the reactive and goal-oriented capabilities of the surrounding BDI system. On the other hand, the choices taken here commit to the truth that there are no homogeneous AI planing systems in practice and theory – known to the author – that would be capable of handling long term and strategical planning in dynamic environments.

5.4 Overview

The design of the planning component is centered around the concept of the domain description and the planning algorithm itself. Figure 5.2 shows a static UML model of the component and its relation to JADEX.

The domain description is created out of the attributes and parameters specified in the ADF `MPlan` element. This is only done if the corresponding plan is of the dynamic type. The description contains all static information needed by the planner. This includes the operators, objects, heuristics, operator instances (actions) and attribute specifications for the objects. The information is extended, every time a new planning process is started.

The operators and object attributes are introduced to the description as a character string. They are stored in form of terms and term instances and used to create actions and retrieve the attribute values from objects at runtime. For this purpose, the description resorts to the symbolic manipulation kernel `JNPU`. It provides a JAVA™ parser and interpreter. In order to prevent multiple occurrences of attribute specifications and operators, the unification and structure comparison procedures are used.

The planning algorithm extends the abstract concept of search and provides it with the goal function (`isSolution()`) and the neighborhood relation (`expand()`). It works on a state structure specified in the concept of a planning `Step`, which derives from `IndexState` and `HashScope` implementing the attribute-value and variable-value mapping functions.

The initial step is created out of the description and the current object-level model. Each following step has a parent, an action, a heuristic estimate and sibling steps associated with. An explicit time attribute spares a variable for time concerned domains. Each step has a goal stack, with the active goal on top. The stack is initialized at the start of the planning process and is modified by the planner and actions.

At runtime the `DynamicPlan` takes the role of extending the domain description with data from dynamic domains and creating a planner instance. It provides the planner with the initial state. Plans extending `DynamicPlan` should implement at least the `setupPlanner()` method and create the initial goal stack. Other methods are used to monitor the progress of the planner and plan execution.

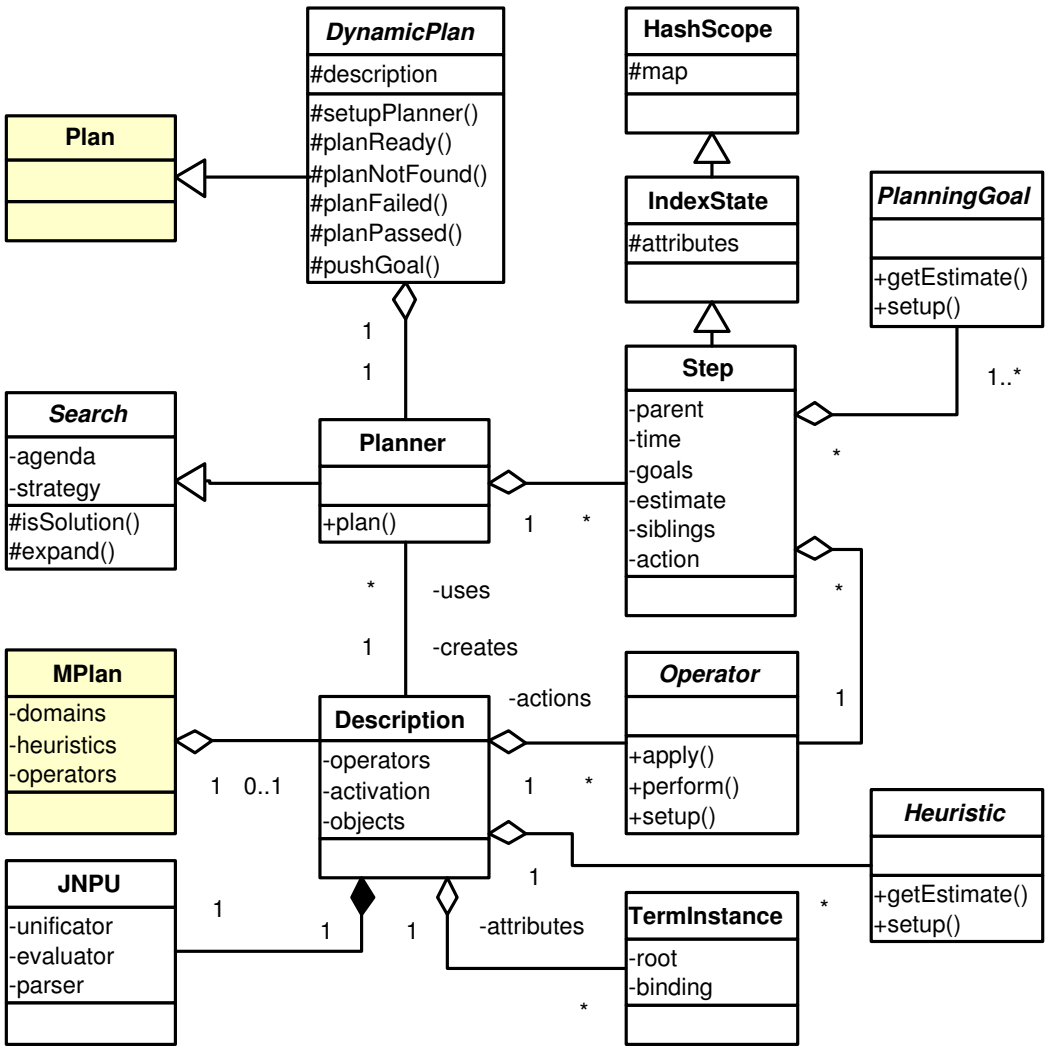

Figure 5.2: Overview: design and integration of the planner. `Plan` and `MPlan` are the runtime and model classes from JADEX.

Operators, heuristics and planning goals require a setup method. It is called after the instantiation, with domain description as an argument. The method can customize the instance with respect to the domain and perform changes to the description. It may also throw an exception in order to inform that the instance is invalid and should be removed.

5.5 Representation of Concepts

For the purpose of planning the application designer has to devise an additional level beyond the knowledge level containing models of objects and processes of the domain. The planner needs a simulation model derived from the objects available for planning. The meta-level model includes relevant object attributes and an abstract description of changes done by processes.

5.5.1 States

There have been several requirements posed on the representation of planning objects state. The planner must evaluate an immense number of possible solutions. In the worst case, this number is exponential in the length of plans created, with the power factor determined by the branching factor of the planning process in the domain. On its way, the number of states is multiplied again by the length of a given solution. The last factor cannot be reduced by control knowledge constraints. These virtual states, called since now meta-level states, need to be easily copied or derived from the predecessor. This poses requirements on the speed of state copy operations and the space a single state can take. Every state can be seen as a simple data base storing variable and object-attribute assignments.

Due to the fact that the planner has to plan with generic, not special crafted objects created by the agent designer in order to hold agent beliefs, it is not possible to work directly with the data modeled within. The planner holds many different and slightly modified views (meta states) in its partial solution agenda. It would be fatal if the views were composed from the original objects contained in agent beliefs. The notion of a meta state aids this problem by copying the state of each object into a meta model that is separated in each view and allows manipulation without conflicts.

The idea of cloning each object in every meta state for the merit of preserving the attributes of the original one would constrain the domain modeler to provide only lightweight objects containing attributes interesting to the planning domain and implement a clone function in each of them. It is not always possible and desired and that is why the planning component provides and works with a generic solution to this problem provided by `IndexState`[3], which is in principle an indexed mapping of object-attribute pairs to corresponding values using hash tables.

In common planning systems based on symbolic representations objects are referred by simple names (i.e. *block*107) and do not carry any information within. To access objects from `IndexState` object references as provided by the JAVA™

[3] `jadex.planning.db.IndexState`

virtual machine may be used. This spares programmer from a solution with virtual IDs. Unfortunately, there is no simple reference mechanism in respect to object attributes, beyond the inefficient `java.reflection` API. The idea presented here allows to access individual attributes by an attribute index number that must be registered at domain creation (cf. sec. 5.8).

The primary meta state used as the start point of the planning process is automatically generated from the beliefs of an agent as stated in the dynamic plan's `<domain>` elements in the ADF. Apart from object attributes, the state also contains the belief name to belief value bindings taken from the belief base.

Each step of the planning process includes a goal stack. The idea here is used in a new genuine way. Instead of one stack of goals that is processed by a planner, every branch of the planning process includes its own goal stack. In this way, the planner is able to simulate the hierarchical decomposition process or aid the planning search with an island approach reducing the planning complexity. A simple ordered hierarchical planner may be simulated by reflecting decompositions as lists of goals and pushing them onto the stack. After a subgoal is fulfilled it is automatically popped up from the stack by the planner and the search may continue towards the next subgoal on the stack. The search finishes, when the goal stack is empty.

5.5.2 Goals

The classical representation of goals assumes a goal to be a set of propositions that must or must not be true in the goal state. Thus goals are simple conjunctions of propositions like $at(robot, position11)$. This is insufficient if the domain contains continuous dimensions and the goal requires the state to fulfill constraints being functions of factors stated upon these dimensions. Again, the goal may be achieved in many ways, mirroring its quite disjunctive nature. At last, there are goals that may require conditions being kept over the whole course of actions and states occurring in the plan.

`IndexGoal`[4] provides an easy facility to implement conjunctive goals representing conditions stated over the attribute-value representation. It is based on the same idea as the meta-level state used for simulation and it is indexed by objects and their attributes.

On the other hand, it is easy to implement all kinds of goals mentioned above, as the code testing for satisfiability is pure JAVA™ and has access to all actions and states included in a plan. This facilitates complex temporal conditions to be stated as goals and the use of full expressive power given by the JAVA™ programming language.

The view of a goal as a function yielding a boolean value reflecting if the goal has been reached or not, is held here to be insufficient. Humans generally use a measure of achievement to guide their actions that is tied down to the goals. This measure may be given by the distance to a particular goal or number of sub-goals to be achieved. It is important to measure the progress of own conduct. In AI

[4] `jadex.planning.IndexGoal`

planning this measure is commonly referred to as heuristic. This thesis introduces an object-oriented view of a goal where the heuristic, being an achievement measure of the goal, is directly provided as one of its methods.

In JADEX goals do not possess an explicit measure of achievement. In achieve goals the target condition describes a boolean function, stating the matter based on agent's beliefs that the goal has been achieved or not. To compare it to the classical planning representation above, an achieve goal is given in the JADEX ADF notation by:

```
<achievegoal name="Go">
   <parameter name="position" class="Point"/>
   <targetcondition>
      $beliefbase.position.distance($goal.position)&lt;5.0
   </targetcondition>
</achievegoal>
```

JADEX allows to state conditions that include continuous factors and functions, but there is no provision to state temporal properties of plans executed for the goal nor is there a way to provide a measure of goal's achievement.

The `jadex.planning.PlanningGoal` permits the domain modeler to encapsulate all this knowledge including the measure of achievement, into the goal. The following code taken from `GoGoal`[5] illustrates the way it is done:

```
public double getEstimate(Step step) {
   if (step.getTime()>deadline) return Double.MAX_VALUE;
   Point spos=(Point)step.get(POS);

   double dx=gpos.getX()-spos.getX();
   double dy=gpos.getY()-spos.getY();
   if (dx<0.0) dx=-dx;
   if (dy<0.0) dy=-dy;

   if (dx+dy<=3.0) return 0.0; // almost there

   return UNIT * (dx+dy);
}
```

The returned measure is the Manhattan distance between goal position `gpos` and current planning state position `spos`. There are two absolute values with special meaning returned by the measure function. `0.0` signals that a given goal has been achieved in the current planning state and corresponds to the `true` value returned by classical and JADEX goal predicates. In the example, it is the case when both points – the destination and current one – are not separated by more than 3 pixels in Manhattan distance.

The `Double.MAX_VALUE` constant is returned if the goal cannot be achieved with the specified plan prefix and the planner should abandon this planning trace

[5] `jadex.examples.storehouse.robot.GoGoal`

at this point. The code above returns this value if the plan prefix extends beyond a specified deadline as stated in the goal. One obvious dilemma with a distance measure is that the values returned by goals and heuristics in a given planning domain must carry the same unit dimension in order to stay comparable.

The reason for melting the heuristic function[6] with the goal predicate lies in the observation that most of the time both rely on the same computation. Thus one function prevents the duplication of code used to model the planning domain.

Apart from the properties described above, modeling goals in an object-oriented way has at least two advantages. First, the goal may not only include attributes describing its properties, like destination or deadline, but it also may contain auxiliary data used by the domain knowledge to store data controlling the search in the planning process. Second advantage is of software engineering type. The goals modeled as class instances can directly derive from each other inheriting attributes and code. Additionally, operators written to cope with one class of goals may be applied to all subclasses of this goal. Further, it is easy for the programmer to create new types of goals assumed necessary for the application. The agents are not restricted to only a bunch of parametrized types provided by the system.

5.5.3 Changes

Changes in the world are modeled as parts of actions. Object-oriented representation uses the concept of method to communicate a change request to the object of concern. This has an advantage for the modeling of processes, because changes denoted by method calls are always local and encapsulate the recipient within the call. Thus, a simple method call is described by an instance of action class belonging to the meta language spanned over the domain model represented by objects and their relations. This allows to provide additional information for the reasoner including abstracted preconditions and postconditions of the given methods. It may include control knowledge used to estimate the applicability of the method in simulated meta states as well.

The change is represented by an operator in the same way as it is done in classical planning. The operator is divided into three parts. The first one is a setup code invoked on every operator so it may register its properties and object attributes it works on. This is done by every instance of the operator. The second part operates on the meta state. It tests the applicability of the operator using control knowledge provided by the domain designer and applies the changes to the meta state. The third part is a direct invocation of the operator on the beliefs of an agent that is done during plan execution.

The code in Figure 5.3 is taken form the `Move` operator of the Blocks–World domain. It represents the control knowledge together with the application code. The first line tests if the goal pursued in the given meta state[7] is of the class `RestackGoal`. Following lines test if the block will be moved to the same location where it is placed. If the block is barred by something else and if the block already

[6] A measure of achievement or distance to the goal.

[7] Named here `Step` because it constitutes a step in a plan.

```
public boolean applyTo(Step st) {
   if (!(st.getGoal() instanceof RestackGoal)) return false;
   RestackGoal goal = (RestackGoal)st.getGoal();
   Block       from = (Block) st.get(DOWN, block);

  // control knowledge
   if(from == to
       || !st.isTrue(CLEAR, block)
       || good_tower(st, goal, block)) return false;

   if(to != null && // the block will be stacked atop another
      (!st.isTrue(CLEAR, to)
        || !to.equals(goal.get(block).down)
        || !good_tower(st, goal, to))) return false;

  // application
   st.set(DOWN, block, to);
   if(from != null) st.setTrue(CLEAR, from);
   if(to != null)   st.setFalse(CLEAR, to);
   return true;
}
```

Figure 5.3: The control knowledge of the `Move($block, $to)` operator.

constitutes a good tower, so it is not wise to unstack it. In any of these cases the operator will not be applied. Further if the destination is not a table (`to!=null`), it must be tested that the destination is clear, that it is the same as required by the goal and the destination must be a good tower that will not be dismantled in the future.

This code obviously interleaves common preconditions found in planning operators with the control knowledge – seen here as an advanced form of a precondition. Such a notion of precondition control knowledge is described by Bacchus & Ady (1999). In the planners controlled by temporal knowledge it must first be analyzed and transformed into precondition control in order to be applied in an efficient way (Doherty & Kvarnström 2001). The solution presented here is more intuitive, because the knowledge is directly provided in the precondition part of an operator.

The application code illustrates two conditional effects. Generally, an operator may contain any type of effects, including conditional and quantified effects, functional temporal relations and even goal manipulation.

In planning systems operators may be stored in form of lifted, not instantiated expressions or they may be partially bound or fully grounded in form of actions. Taken the fact that JAVA™ does not allow any partially instantiated objects, the operators may be stored as a class reference, a string describing their constructor or as an instance of this class. The last solution is used here. There are several advantages to have all operators fully instantiated.

First, the constructor may signal that the variable binding presented to it do not make sense by throwing an exception and prevent a creation of an action. Second,

the planner knows the number of available actions in advance and does not need to access the interpreter to instantiate new operators at the simulation and planning time. Third, all actions may be referred by JAVA™ technical reference, which offers a better performance.

On the other hand, the number of parameters or variables an operator can be bound to is limited. The number of actions that must be created and tested at the domain initialization phase, is of $O(o \cdot i^p)$ where o is the number of operators, i – the number of objects in the domain and p – the maximum number of parameters for an operator.

Special care must be taken not to equip operators with too many parameters. I.e. the operator `Move($b, $from, $to)` can also be modeled as `Move($b, $to)`, because in the forward state space search it is always known where the block `$b` is placed on. One could also split the operator into `Pickup($b)` and `PutDown($at)` in order to further reduce the number of actions.

5.6 Symbolic Processing

This section describes the design of term processing tools used for symbol manipulation and reasoning by the planner. It describes the basic fundamental representation of terms, the parser transforming JAVA™ expressions into such terms and a bunch of other tools used to compare and reason with them.

5.6.1 Term Representation

The terms used to represent expressions from the processed language are stored in tree structures. There are different kinds of nodes representing these structure elements. Every `Node` has an interface requiring it to have a *type* and optional *child nodes*.

The type attribute is used as a form of run time type identification. It bypasses JAVA™'s own type system i.e. the type of a node is different from its JAVA™ class. Using an explicit type attribute greatly speeds up the node processing allowing the use of `switch` constructs. This solution is responsible for less than 10 classes implementing the `Node` interface.

Most terms created by the parser and other term processors do have a final, not mutable character. This allows to store a fixed number of child nodes in their parent node. The convenient way to store these child nodes is to put them into an array. `Leaf` nodes representing final nodes in a tree do not allocate such an array at all.

The term constructors are placed, away from the node representation, into a separate `NodeFactory` containing a factory method for every node type. Any other term sub-processor implements some functionality separate from the actual node representation. This design diverges from the object-oriented programing paradigm in favor of flexibility, clarity and scalability.

5.6.2 Java Parser

The parser for right hand expressions of JAVA™ and the OQL-like language has been developed using JAVACC™ – a Java Compiler Compiler[8]. JAVACC™ is a generator for top down LL(1) parsers. The ability to include token, syntactic and semantic lookaheads allows to extend the languages available for parsing to the LL(k) language family. At the input of language grammar and lexical pattern specification JAVACC™ generates JAVA™ code output including the parser and a lexical analyzer.

In order to build up syntactic trees like the *term structures* described above, the programmer has to interleave the grammar with semantic actions being in principle JAVA™ code that creates the trees. There are at least two tools for JAVACC™ to do this automatically. JJTREE comes with the compiler compiler and has been used to produce syntax trees for the general JADEX system. Unfortunately, the tree structures produced by JADEX parser were not designed for the purpose of symbolic manipulation including unification and structure transformation, which is required in the process of reasoning. JADEX structures include code, both for a precompiler and for an interpreter used by the JADEX system and therefore they bear much more weight than the *term representation* described above.

JTB[9] is a separate solution that creates syntax trees using the *Visitor Pattern* for manipulation. Although this tree generator tool seams to provide an elegant solution, after some time experimenting with it, the author turned to a much simpler own approach. The *Visitor Pattern* used by JTB has two disadvantages. First, with the growth of processing functions, predicates and procedures that may be applied to the tree nodes, the node interface grows – requiring an entry for each type of processor and number of arguments. On the other side, with each node type all visitors and their interfaces must be enhanced by procedures handling this node. Second, it is quite inefficient to use this visitor pattern, as it requires at least two virtual calls – one to the node and one back to the visitor – that cannot be well optimized by a compiler and generally take hundreds of times longer than a simple `switch` construct.

The code generating *term structures* is directly placed among the grammar rules provided to JAVACC™. This code heavily uses help classes like `ParserUtil`[10] and a node factory in order to reduce the size of the grammar file and keep grammar rules clean. The lexical analyzer `Token`s are capable of parsing long, hexadecimal and octal numbers as well as unicode escape characters.

This approach yields quite lightweight syntax tree structures that are used by components like precompiler, interpreter, unificator, normal form transformer, substitution and structure comparison.

[8] `https://javacc.dev.java.net/`
[9] Java Tree Builder: `http://compilers.cs.ucla.edu/jtb/`
[10] `jadex.planning.java.parser.ParserUtil`

5.6.3 Unification

Unification is a basic operation working on a set of terms. Speaking informal two terms unify if they are similar in their structure and their variable bindings correspond to each other. Unification may be used i.a. for theorem proving or pattern matching.

The unification algorithm used here owes greatly to the *linear unification algorithm* by Henckel (1993). The data structure manipulated by it is an array of `VarBinding`s containing the variable binding for every variable of a given term. The variables of a term can be bound to a term instance being basically an aggregate of a term and the related binding. It can be bound to a chain of variables where all the variables are unified, or it can be bound to a JAVA™ `Object`, which is a shortcut saving a node wrapper around this Object.

Given two term instances defined by term roots and variable bindings the algorithm returns true if the term instances unify or false in the other case. As a side effect this algorithm changes the variable bindings of both term instances. Please see Appendix A.1 for the corresponding pseudo code.

5.6.4 Interpreter

The JAVA™ interpreter `SEvaluator`[11] derives from a simple node processor. It extends the processor with the ability to evaluate terms with variable bindings and within a scope of evaluation. The last provides bindings for variables that are unbound in a given term instance.

The processor contains a simple aggregate value A called *accumulator*. The code working on this accumulator is a simple `switch` construct that takes the input term structures and creates in a bottom up manner a value. It is guided by the types of term nodes. The code below illustrates this approach:

```
protected void process(Node node) {
   [....]
   switch(node.type()) {
   case CHOICE:
      process(n[0]);
      if (A.isTrue()) process(n[1]);
      else            process(n[2]);
      break;
    case NOT:
      process(n[0]);
      A.negate();
      break;
   [....]
}
```

The `CHOICE` command corresponds to the JAVA™ expression `a?b:c`. If a is true return the result of b and in the other case c. The `NOT` command corresponds to

[11] `jadex.planning.java.npu.SEvaluator`

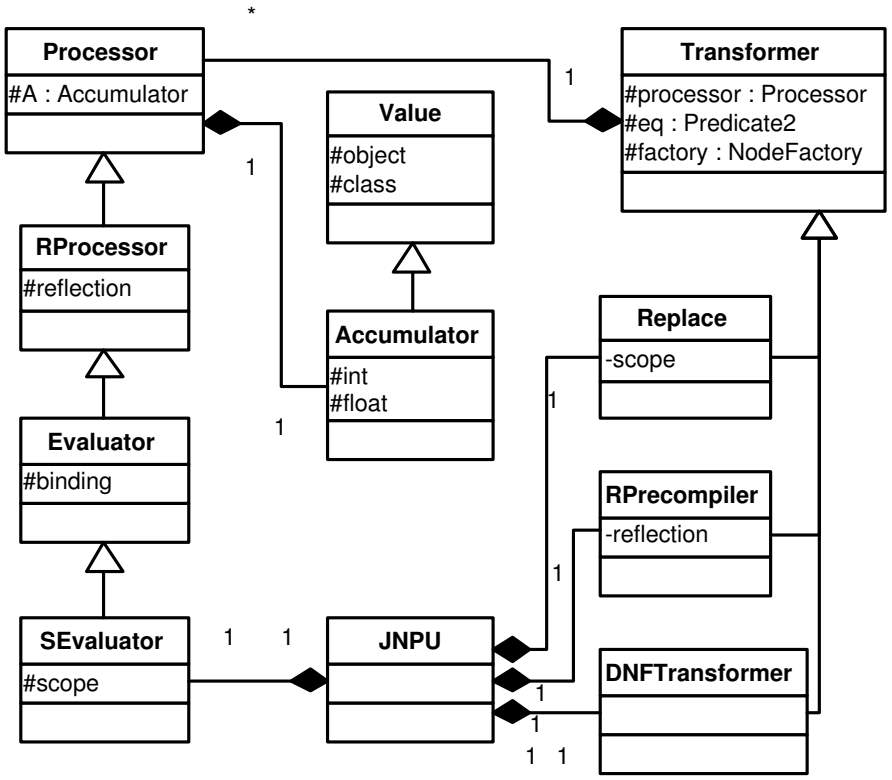

Figure 5.4: Term processors and transformers and their relation to each other.

boolean negation. The interpreter simply asks the accumulator to negate its value. The children of a term node are stored in an array accessed by their index. The `CHOICE` node has three children, the `NOT` node only one. The use of a single value as an accumulator has the advantage that the fundamental process of evaluation does not produce new objects for each operation and thus, stays less memory intensive.

In case of some variable being undefined or if an expression does not make any sense, the `VoidValueException` will be thrown in any part of the interpreter code to indicate to the structures above that a sub-expression below yields no value. The `VOID` value represents no value, like returned by a void function. Normally, a JAVA™ programmer does not need to care about something nonexistent, but in an interpreted program it is useful to handle and reason with undefined entities in an appropriate way.

Most of the other procedures and predicates defined over term structures use a kind of processor to evaluate and compare parts of this structures. This is illustrated in Figure 5.4. The `Accumulator` is a modifiable `Value` that holds float and integer basic registers to operate on them. The `Processor` operates on a single `Accumulator`. `RProcessor` is a processor that may access JAVA™ objects using reflection. The `Evaluator` possesses the ability to use values bound to variables of a term and `SEvaluator` lookups the values of variables in a scope in the case they are not bound in the term instance.

The `Transformer` is a basic super class of all node transforming utilities. It

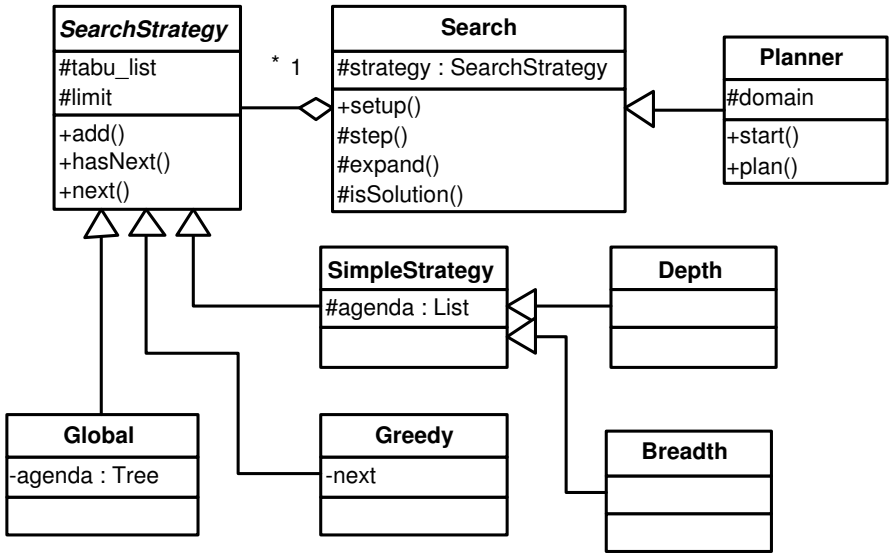

Figure 5.5: The search algorithm and its relation to search strategies and the state space planner.

uses a processor to retrieve the value of a sub-term, an equality predicate to compare terms based on their structure and a node factory to produce new nodes. `Replace` is a simple substitution on variables given by bindings defined in a scope. `RPrecompiler` analyzes the term structures and produces simplified ones. It uses JAVA™ reflection to access object values. `DNFTransformer` produces a disjunctive normal form of JAVA™ expressions if it is possible.

All of the tools working with term structures and instances including a parser and a pretty printer are composed into a single unit called `NPU`[12]. It owns to the Facade Pattern[13] and presents a simple functional interface to all of the tools mentioned. For the JAVA™ language subset supported here, the instance in use is of the type `JNPU`.

5.7 Search Algorithm

The search is a functional wrapper around the search strategies. It allows to select and setup the strategies and includes common code performing the search. `Search`[14] is an abstract class thought to be extended. A single method called `step()` should be called repeatedly until it returns a solution. To determine if an object is a solution, the deriving sub-class needs to implement the virtual method `isSolution(Object obj)`. The search must also be given the neighborhood relation defined over the object of the search domain. This is done with the virtual method `expand(Object obj)`, which returns all neighbors corresponding to a given object.

[12] An acronym for Node Processing Unit.
[13] Gamma et al. (1995)
[14] `jadex.planning.search.Search`

Figure 5.5 illustrates the relation of `Search` to the strategies. Any search can be performed using the `Greedy, Global, Depth` and `Breadth` strategy. The difference among them is exemplified by the number of partial solutions considered and the order they are chosen to be expanded. `Greedy` search only considers one object at a time. `Depth` and `Breadth` store the objects in a double linked list that is accessed at the beginning or the end. `Global` search stores objects in a set implemented using a balanced tree. The objects are sorted with help of a comparator.

The search algorithm asks the strategy for the best partial solution so far. Then it tests if the specified object is a solution. In this case it will be returned. Further, the neighborhood of this partial solution is presented to the strategy and the search may be continued. If there is no object left to be expanded, the `step()` method throws an exception. The code for this function is given below:

```
protected Object step() throws SearchException {
    if (strategy.hasNext()) {
      Object head = strategy.next();
      if (isSolution(head)) return head;
      Object[] neighbours = expand(head);
      strategy.addAll(neighbours);

      return null;
    }
    throw new SearchException("Search reached a dead-end.");
}
```

In its basic form the search may be given a set of taboo states. It is common to use a taboo list or visited state list by search algorithms to prevent cycles in the search space. With an abstract concept of a taboo set the user may specify any regions of the search space that should be avoided. This set is expanded in the process of search by states visited. Again, the user may choose what to do with new visited states added to the taboo set.

5.8 Domain Description

Any planning process needs a description of the environment where the plans will be executed. It includes a number of objects in the domain and actions used to manipulate them. The choice of a heuristic planner allows to state in the domain descriptions any heuristics that would help the planning process. These generic heuristics may be seen in the context of this thesis as optimization criteria posed over the planning problem. Heuristics may also be used as safety and liveness conditions to be enforced over the plans generated.

The operators are presented to the domain description as a string indicating the constructor of the operator class. The invocation includes variables denoted by `$` that may be bound to any object. Every time a new operator is added to the domain description, actions are instantiated using the constructor and specification of variables, which are bound to any object already present in the domain including

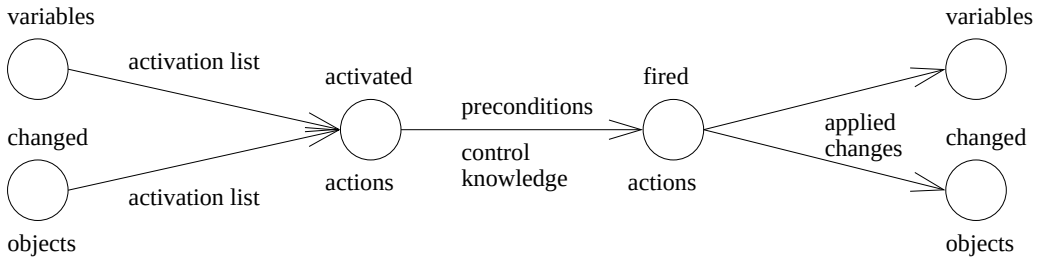

Figure 5.6: Actions are activated by changes in objects or variables and if they are applied to the current state, induce changes in variables and objects considered in following states.

the `null` value. On the other hand if objects are introduced into the domain description, they are tried as arguments to the constructors and used to create new actions.

Activation of actions. The domain contains a mapping from objects and variables to a lists of actions used further by the planner to optimize the choice of operators at each planning step. Actions subscribe to objects and variables they are interested in and are activated whenever object attributes change or the variable is bound to another value. By default an action should subscribe to all objects and variables it uses in its preconditions. Actions can also explicitly inform the search that conditions regarding an object or variable have been changed. An action can also request to be always activated, independent of the conditions in the domain.

The activation change cycle is illustrated in Figure 5.6. The actions are activated by changes to objects and variables where they subscribed to. In the next step the preconditions and domain knowledge are applied to filter out actions not applicable in the current state of simulation. Thereafter a new simulation step is created and effects of actions are applied to it. Every step has annotated the changed objects and variables in order to activate actions in following steps.

Static vs. dynamic domains. The number of operators and heuristics may be stated in advance and remains constant through the time line of the domain description, which is the life time of an agent. On the other hand, there are domains where objects do not retain their original state and where the number of objects varies with time. It is a natural application for a planner to operate in domains where object attributes change. If objects would not change, a plan could be devised to suit the domain all the time.

Most planners start with a description of a domain that is static and generate a plan for it. The term dynamic domain refers to the ability of planners to reason with changing number of objects generated or destroyed at the planning time. Considering this convention, the domain descriptions presented here may only contain semi-dynamic sub-domains. The feature to handle changing numbers of objects at the planning time can be simulated using variables in the same efficient way like the use of changing resources.

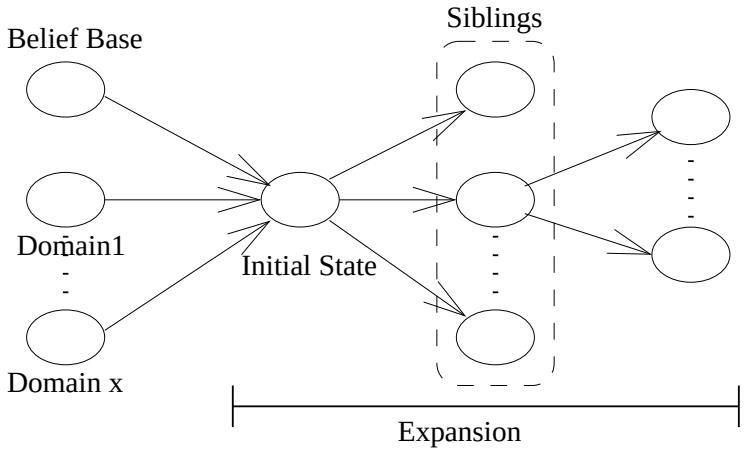

Figure 5.7: The initial state is created from the belief base of an agent and initialized with objects from specified sub-domains. The planner performs expansion on states taking into account activated actions and actions from states created in previous expansion (siblings). The arrows indicate flow of information.

The term dynamic domain refers here to the fact that the planner needs to create plans again and again while its beliefs change. Therefore, the domain description may contain subsets of objects that change each time the planner is asked to create a new plan. For each of such subsets the domain description is regenerated. This is done by cloning all static parts of the domain description and inserting objects from dynamic parts. Clearly if all object subsets are marked static, there is no need to instantiate new operators and update activation lists (described above), which yields a performance advantage.

5.9 Planner

As can be seen in Figure 5.5, the planner derives directly from the `Search` and reflects in this way the fact that it is a form of search. The only special method is the state expansion procedure. All other functions are simple wrappers around the methods of the search. In the setup method the planner does not provide the search with a taboo set. This is done intentionally, as for most planning spaces it is difficult to define a generic notion of equality[15]. In the `plan` method the solution is returned in form of a reversed order vector containing all steps of the plan.

The expansion procedure is illustrated in Figure 5.7. The initial planning state (initial step) is created out of descriptions of sub-domains and the current variable binding. The planner expands the first step trying all actions for their applicability. The next step is expanded in a more elaborate manner. First, all actions are tried that have been successful in the previous state. This actions created sibling states and it is assumed that their preconditions and control knowledge will still render them applicable in the current one. For this purpose every state – after an action

[15]"On those stepping into rivers staying the same other and other waters flow." (Heraclitus, 600BCE)

has been applied to it – is augmented by an array of states created in the same expansion phase. After this array has accomplished its duty, it is erased in order to reduce the inter-linkage of object references in the search space and improve garbage collection. Second, actions are tried that registered their interests in object or variables changed in the previous state. The planner simply asks the domain description element for this activation relation, which is stored there in an explicit manner. Third, there are actions, which are always activated. They are tried at last.

A new step is added to the planning search agenda after an action is successfully applied to the previous one. However, before they are included into further search, heuristics are used to rate the new step. If a step fulfills a goal, as indicated by the goal achievement measure, this goal is removed from the goal stack of that step. All new steps that survived this expansion process, are collected in an array of siblings that is later referenced by each of these steps.

5.10 Integration with JADEX

The planner is integrated into JADEX in a bottom-up approach using the monitoring and replanning approach. It follows the ideas and argumentation presented above and motivates an integration design, in which the planner takes over the domain specific reasoning part and the BDI system performs control, monitoring and reactive tasks.

The integration requires an extension to the model and runtime parts of the JADEX core system. In order to reduce interference only the `MPlan`[16] element of the model is modified. The runtime features of this extension are placed in a sub-class of `Plan`[17] called `DynamicPlan` to reflect its dynamic nature.

5.10.1 Model

The Agent Description File allows for flexible definition of agents and their properties. This advantage can be used for the specification of planning domains and problems. The XML representation of a dynamic plan is an extension to the JADEX representation of static plans stored in the model. Figure 5.8 illustrates this conduct. The model element for static plans is augmented by the sub-elements: domain, heuristic and operator.

With the *domain* element the user may specify several sub-domains of objects used by the planner. This objects may come from beliefs, goal parameters or plan variable bindings. They are used in the domain description element to instantiate actions. With the *static* attribute set to **true** in the ADF element it is stated that objects in this sub-domain do not change in attributes and number in a degree that would require to regenerate actions concerning these objects.

The *heuristic* element specifies any heuristics that may be used to aid the search. The content is a string that describes the constructor. Heuristics are instantiated

[16] `jadex.model.MPlan`
[17] `jadex.runtime.Plan`

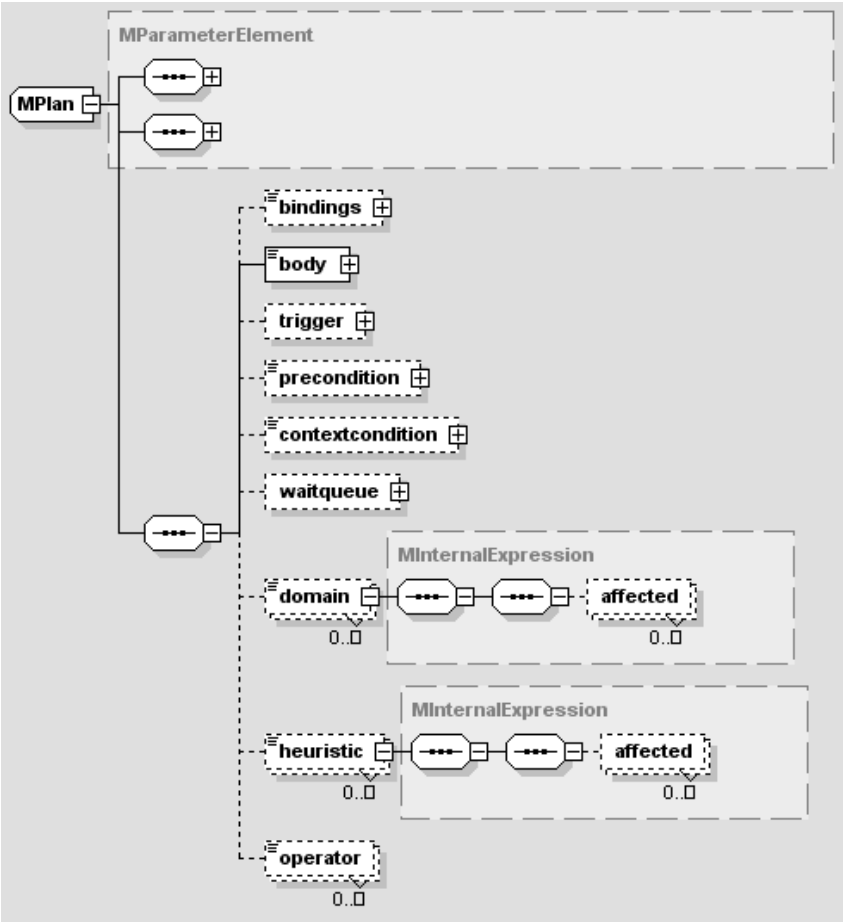

Figure 5.8: The plan concept form the Agent Description File notation is extended by the features: domain, heuristic and operator.

the first time a dynamic plan of this kind has been created and are used further for any successive dynamic plans.

With the *operator* element any operators are specified that should be used by dynamic plans. The content describes the constructor of the operator as a lifted expression. Actions in domains, which contain dynamic sub-domains, are instantiated every time the dynamic plan instance is created.

Options. There are three options controlling the simple search algorithm of the planner (cf. fig. 5.9). These options are reflected in the extended attributes of the ADF `plan` element.

The choice of search strategy determines directly the trade-off between speed and plan quality. Only two of the four strategies are interesting for the planner. The depth first search places emphasis on the speed of planning and is useful for problems like the blocks-world domain with many acceptable solutions. The agenda limit determines the backtracking memory for the planner and may be chosen to be arbitrary large, as it only determines the space requirements of the algorithm, but not the speed. For complex problems with few solutions the agenda limit parameter

determines the completeness factor of the depth first algorithm.

The global search strategy is different from local strategies like depth first search. This fairly flexible strategy allows to control the trade-off between speed and optimality using the size of agenda. The speed seems to be proportional in the size of agenda for domains like blocks-world.

strategy		
	greedy	Uses a form of depth first search, but without an agenda and backtracking.
	depth	Appends the nodes from the current neighborhood to the end of agenda after sorting them in the order based on their heuristic cost.
	breadth	Like above, but the nodes are perpended.
	global	The nodes are inserted into the agenda in the ascending order based on their heuristic cost.
agenda	1–N	agenda is the list of states that are remembered by the search to be expanded when they become interesting. The list is limited by the memory constraints of the system. The nodes at the end are pushed out of the agenda if the limit is reached. Setting this to 1 forces greedy search.
time-limit	0–N	specifies the limit of milliseconds that should be used for planning. If this limit is reached, the planner fails to find a plan.

Figure 5.9: Planner Options

The model embodies a planning domain description constructed out of the elements and parameters given in the ADF. This domain description is created the first time a dynamic plan instance requests it. The description contains all operators and static domains. All actions instantiated from operators using objects contained in static domains are also included. The heuristics are handled as static elements and instantiated only once.

5.10.2 Runtime

In order to come into favor of planning, the plans stated in ADF must derive their body form the class `DynamicPlan`. The dynamic plan has been designed as a normal Jadex plan going through two phases. The first being the planning phase, the second being the execution phase.

At runtime a clone of the domain description is generated and populated with objects from the dynamic sub-domains, if any. Then an instance of the planning processes is created. This new planner requires a setup code where, among others, the goals and timing constraints may be stated.

The initial state is created each time and appended to the planning agenda. The variable binding of the initial state is a direct projection from the beliefs of an agent. The names of beliefs and belief-sets become variable names in the initial state. The values of beliefs are directly mapped to the values bound to variables. Belief-sets are represented as object arrays.

At this point the planning is started to produce a new dynamic plan in form of a sequence of actions. The success of this phase is signaled by the method `planReady()`, the failure by the method `planNotFound()`. The second phase executes the sequence step by step by calling the `perform()` method of actions contained in the plan created. If any of these actions signals a failure by throwing an exception or returning **false** as result, the plan is regarded a failure and the method `planFailed` is called. The method `planPassed()` signals the success of the execution phase.

Replanning. The JADEX system extends the PRS paradigm and inherits good characteristics in respect to reactivity and control of autonomous systems. This allows to see the planner as a mere extension to the architecture. In this notion, the JADEX system is given the responsibility to monitor the execution of a dynamic plan and activate replanning on failure.

The dynamic plan may be seen as a set of many or even countless number of plans that may be created by the planner. To achieve replanning the JADEX system must not exclude the dynamic plan every time it passes. This is easily done with the BDI flag `exclude="never"`.

Regrettably, there is no way for a plan to modify its proprietary goal. The system lets every plan to work on a new instance of `ProcessGoal` and prevents intentionally any conflicting changes to the original one. This way no plan can save state information regarding the goal anywhere else but in the belief base. Also, there is no other way for a plan to state that it is not applicable for a goal apart from the mechanism provided in the ADF using `<precondition>` and `<contextcondition>`.

The dilemma occurs, when there is no possible plan for the goal that can be constructed by the planner. To ensure that the planner is not asked again and again for an impossible thing, one must state `exclude="when_failed"` and let the `DynamicPlan` fail. This still allows for replanning. Taking the notion of `DynamicPlan` as a set of plans that are generated at runtime, it only fails if such a set becomes empty at any point of the time. In the other case, when an element of this set simply does not achieve the requirements of the pursued goal, the planner will be called again for the same purpose.

5.11 Summary

This chapter justified and presented a design of a state space planner the has been integrated into a BDI system. The approach taken here abandons the long tradition in artificial intelligence to incorporate the planner in an agent system as a central component influencing the whole architecture. This result is mainly due to the fact

that this thesis started with a complete BDI system and the only reasonable way to attach the planner was to compose it into JADEX as an extension.

Other systems followed this way, but this thesis is the first one known to the author to incorporate a state space planner based on recent results showing benefits of planning using control knowledge. Again, the planner designed goes even further applying the knowledge in form of precondition control. Also, it is only one planner known to the author that is able to fully utilize the JAVA™ representation for actions and goals and that reasons with JAVA™ objects as primary concepts of the planning domain.

It has been shown that it is easy to extend JADEX with AI planning concepts and – to speak generally – with any reasoning and problem solving mechanisms. The hybrid approach presented here benefits from both: the symbolic manipulation and envisioning capabilities of the AI planner, and reactive, goal-oriented and effective procedural reasoning techniques of the BDI system.

Chapter 6

Evaluation

The design and evaluation of the planner component was guided by the blocks-world domain. For further evaluation a dynamic planning domain was created in the shape of the Dock Worker domain. Both domains are described here and results regarding the performance and usability of the planner component under the control of the JADEX system are presented.

6.1 The Blocks

The blocks-world domain is a standard testing domain for all planners. It was one of the first problems investigated with planners and from the beginning it posed a challenge as the problem is clearly exponential.

The first prototype of a partial order planner was able to stack more than three blocks, what was assumed in the beginning of this diploma project to be a good result corresponding to same achievements of other partial order planners. Inspecting the space of plans produced, it was immediately clear that this approach is too complicated in the design and workings, and there surely exist techniques able to solve such problems in a better way.

Based on the control knowledge borrowed from the $\mathcal{TAL}$PLANNER and adapted to the JAVA™ notation, this state space planner is able to stack as many as 500 blocks. Using global search with an agenda of 10, it requires about 10 seconds on an i586 400MHz machine to stack 100 of blocks and about 30 seconds for more than 200 blocks. Using depth first search and agenda size of 100, it stacks 500 blocks in 5 minutes on the same machine.

The speed of planning could be increased further. A pure domain specific planner, spilling last drops of generality, should be "much, much more faster", but for this design the result is quite comparable, because the planner copes in a run using 500 blocks with about 250000 grounded action instances. This quantity could be reduced by splicing the `Move($b, $to)` operator into `Pickup($block)` and `PutDown($at)` operators.

GUI. The user interface to the program is kept simple. There are two views, one of the current situation and one of the target situation. There are three buttons.

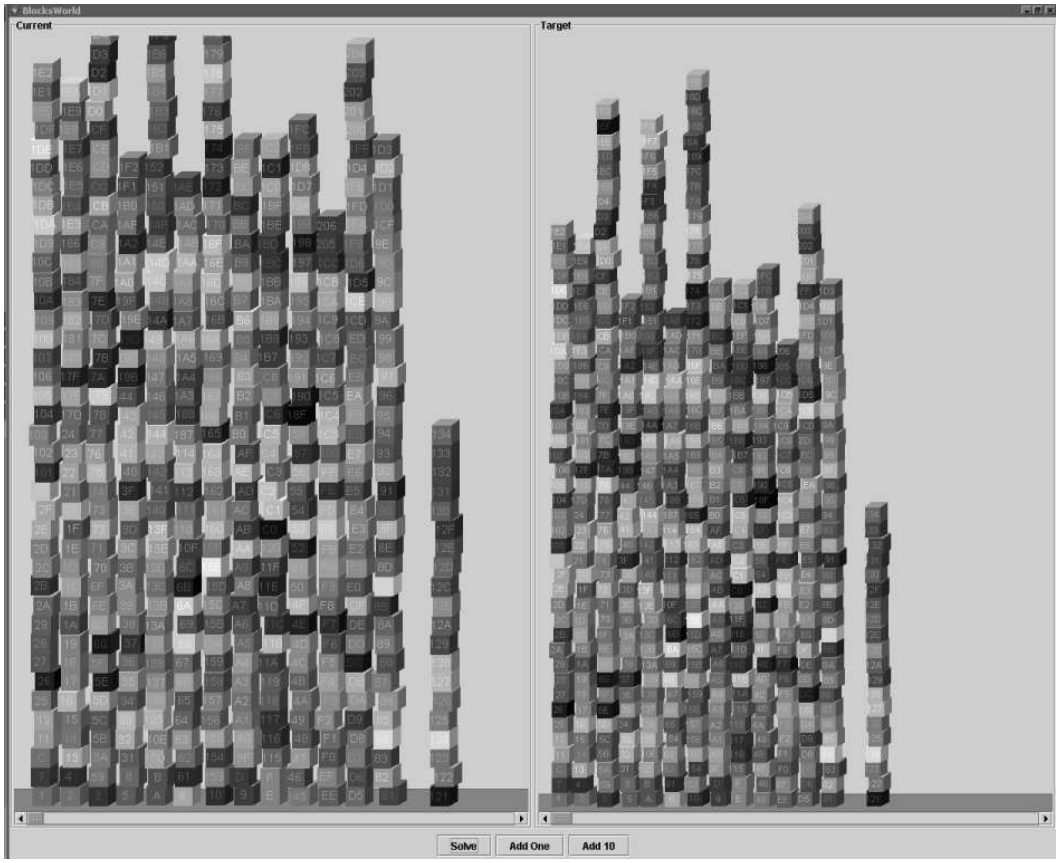

Figure 6.1: Blocks-world GUI. In this view the planner just coped with a problem containing about 500 blocks.

The first is used to pose the restack goal. The second to add a single block of chosen color and the third to add ten random blocks (cf. fig. 6.1). The last button was added after the planner has been successfully tested with 50 of blocks, but the user is able to add more blocks at once by making a double, triple or quadruple click onto the canvas. The number of blocks added is a square of the number of clicks. In order to manage the situations, blocks in the current and target view may be moved with a mouse and dropped onto stacks and different positions on the table.

Domain. The domain is modeled as a simple list of `Block` objects having two attributes. The `CLEAR` attribute says that there is no block above the one in focus. The `DOWN` attribute gives a reference to a block below or `null` if the block lies on the table in the current simulation state.

The goal has an array of blocks that must be considered. It signals with estimate value of 0.0 that all blocks are placed on their destination positions. In the other case, it returns the number of blocks that will certainly have to be moved. This is done using the `bad_tower` heuristic yielding the number of bad blocks in a tower by simply counting all blocks from the top of each tower that do not fulfill the requirements of the goal.

There are two settings. In the first one only the operator `Move($block, $to)`

```
public boolean applyTo(Step st)  {
   if (st.get(LOAD)!=null) return false;
   if (!(st.getGoal() instanceof RestackGoal)) return false;

   if(!st.isTrue(CLEAR, block)) return false;
   if(good_tower(st, st.getGoal(), block)) return false;

   st.set(LOAD, block);
   st.set(DOWN, block, GRIPPER);

   Block from = (Block)st.get(DOWN, block);
   if(from != null) st.setTrue(CLEAR, from);
   return true;
}
```

Figure 6.2: Control Knowledge for `Pickup($block)`

is used. This operator is activated whenever the `$block` changed its position or something happened to the destination. This activation on change reduces the number of actions tested against a simulation step to a logarithmic quantity. Given 100000 action instances only about 60 will be activated. The control knowledge of this operator has been described in the chapter before (cf. fig. 5.3).

The optimization heuristic used is `PlanLengthHeuristic`[1] that might accept a weight argument. This weight factor is used to make the search to perform goal- and speed-oriented (<<1.0) or to focus on quality (=1.0). The heuristic may be seen as a form of optimization metrics.

The planner is activated by a dynamic plan called `RestackPlan`[2]. The only function of it is to take over the JADEX goal and instantiate the `RestackGoal` used by the planner.

The whole design is described in the ADF by the following XML code:

```
<plan name="restack" strategy="depth" agenda="100">
   <body>new RestackPlan()</body>
   <trigger><goal ref="restack"/></trigger>

   <domain>$beliefbase.current</domain>
   <heuristic>PlanLengthHeuristic(0.5)</heuristic>
   <operator>new Move($block, $to)</operator>
</plan>
```

Here the planner is asked to use depth first strategy with a backtracking memory of 100. The JADEX goal is called *"restack"* and has as parameter the target configuration of the table. The domain includes all blocks from a belief describing the *current* situation, which is marked as dynamic sub-domain by default.

The second setting uses two operators. The `Pickup($block)` operator is de-

[1] `jadex.planning.PlanLengthHeuristic`
[2] `jadex.examples.bw2.RestackPlan`

```
public boolean applyTo(Step st)  {
  Block block=(Block)st.get(LOAD);
  if (block==null) return false;
  if (!(st.getGoal() instanceof RestackGoal)) return false;

  if(at == null) { // table
    Step prev=st.getPrevious();
    while(prev!=null) {
      if (prev.get(DOWN, block)==null) return false;
      prev = prev.getPrevious();
    }
  } else { // other block
    if (!st.isTrue(CLEAR, at) ||
        !at.equals(goal.get(block).down) ||
        !good_tower(st, st.getGoal(), at)) return false;
  }

  st.set(DOWN, block, at);
  st.set(LOAD, null);
  if(at != null)   st.setFalse(CLEAR, at);
  return true;
}
```

Figure 6.3: Control Knowledge for `PutDown($at)`

rived from the first part of the move operator. It picks up the block and puts it into a gripper. Then it modifies the *LOAD* belief of the planner agent stating to what the agent is holding. In order for this operator to succeed, the agent should not carry any block at that time. The block to be lifted must be clear and the agent should not dismantle any towers of blocks, which are already in the goal configuration (cf. fig. 6.2).

The `PutDown($at)` operator places the block loaded by the agent at the specified position and sets the *LOAD* belief to `null` in order to indicate that the agent is no more holding a block. For success of this operator it is required that there is actually a load that may be placed anywhere. If the block is going to the table, it should be assured, it is only done once[3]. Otherwise, the destination block should be clear, reflect the expected goal situation and it must be on top of a good tower (cf. fig. 6.3).

The last setting suffers from performance losses when compared to the first one. The split in two operators causes the branching factor to multiply for each of the operators spreading the search space far more than expected. The second cause of this negative effect is due to the fact that the activation-change process cannot be used in such an effective way like in the first setting. The activation for the block picked up and the block freed extinguishes after the put-down operator has been applied, and cannot be used for pickup operators. This requires all operators

[3]This is a nice example of using a temporal safety condition in form of a JAVA™ `while` loop.

to be activated at any time of the planning process. It is an unsatisfactory result as it shows that the efficiency of this planner concept is sensitive in the choice of operators and control knowledge used to solve the problems.

6.2 Loader Dock

The Loader Dock example domain consists of two kinds of agents. There is a single store agent responsible for initializing the graphical presentation, simulation of storehouse processes and initialization of other agents. Worker agents, on the other hand, take over the planning and are responsible for carrying packets around the store. The story is following: in a storehouse there are several workers wandering around and carrying packets between incoming trucks and shelves. Their job is to unload packets from trucks coming in with load, or to deliver packets to trucks arriving with no load at the store. The store itself contains several shelves where packets can be temporally placed. The shelves are separated by corridor ways, which are used by workers to transport the packets.

This particular domain has the specific properties[4] of being:

- Fully observable in respect to a certain update interval of mutual beliefs that is performed by the store agent.
- Almost deterministic. The agents cannot be sure if the plans created will be timely executed as they have predicted.
- Dynamic. The environment changes due to other agents and processes simulated by the store agent. This includes trucks coming in and going at various time intervals.
- Continuous. Most quantities like packets, trucks, robots and places are finite, but the attributes of these objects like speed, direction, arrival or departure time are real valued.
- Concurrent. There are many processes and agents acting in this domain concurrently.
- Cooperative. Agents do have common goals to handle the job at the storehouse, but must share resources like time and corridor space.

Storehouse model. The storehouse depicted in Figure 6.4 is modeled using a discrete grid of points connected with each other through pathways. Each point can have many types. There are obstacle, junction, store, load and start points. To prevent robots from driving above shelves, walls and dock areas, obstacles in form of rectangles are defined for each such object. The positions and widths of objects are stated in real values.

The model includes three dock areas with six load points each. These dock areas are used by trucks coming in with packets or taking packets out. There are

[4] Cf. the list in Section 2.2.

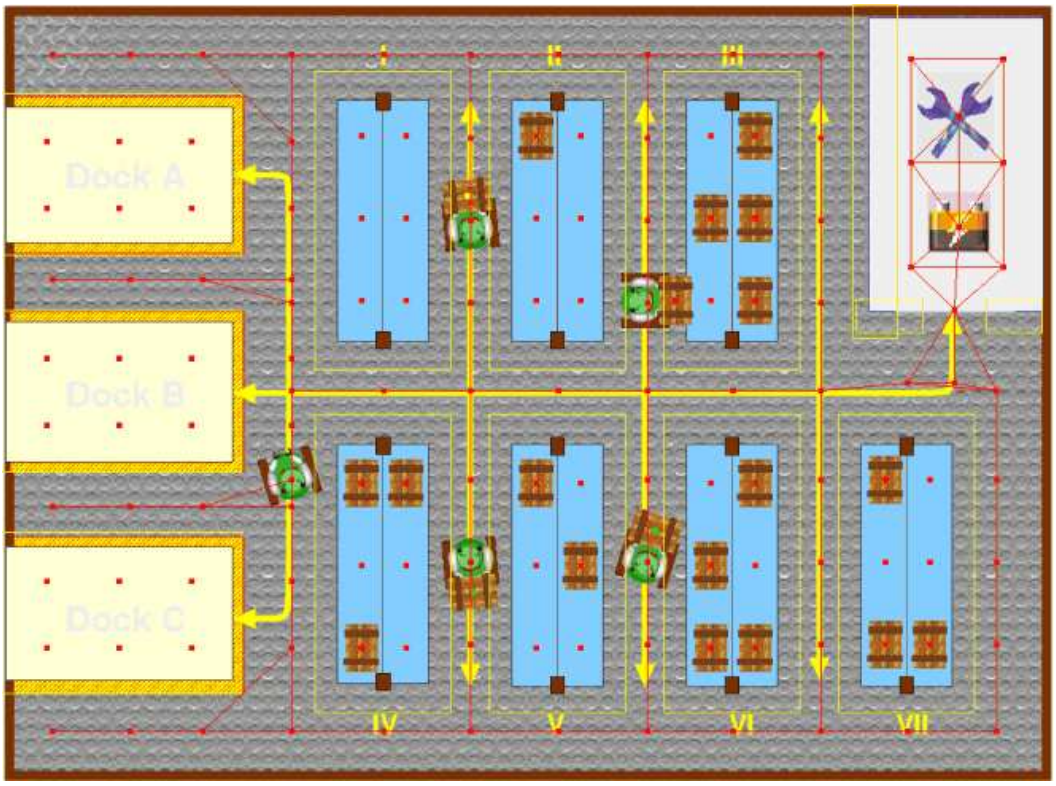

Figure 6.4: Storehouse floor plan overlaid with the way point reticle. The points and pathways are marked red (black). The obstacles are marked as yellow (white) rectangles.

seven shelves with six store points each, used to place packets for temporal storage. A robot park area is present where the robots take their starting positions after being initialized by the store agent. The repair and load stations are reserved for further domain extensions.

The example domain may include up to six worker robots. Each robot is represented by a sprite made of 64 left and 64 right images depicting it in different angles and steps for animation purposes. The same is true for the packets that additionally have a color marker atop. Trucks are illustrated as simple rectangles that may carry loaded packets. The robots and packets can be positioned anywhere in the store and include a real valued direction attribute reflecting their azimuth from the north heading.

Moving around. Most plans and goals of a worker involve movements around the store. The `GoGoal` is a superclass of many other goals like go away, pickup or put-down goals. It contains as an attribute the position where the agent should go and the deadline, by which the agent should arrive there. For the aid of control knowledge the goal also stores a mapping of points to the fastest way connecting them to the start position. This mapping prevents the planner to consider paths that are certainly worse than already tried.

The operator applied in this planning problem is `Move($to)`[5]. It registers its interest in all points surrounding the destination specified by the `$to` argument to the constructor. It will be activated any time the robot stands near such a point. The control knowledge is more complicated than in the case of blocks-world operators. First, it must be assured that there is no static obstacle between the start position and destination. Second, the operator needs to compute the distance to move over and the change in the heading of the robot if it needs to turn. Based on this values the time needed for this action to be performed is computed. Given the starting point and end point, starting time and finish time, the operator checks this data against a dynamic model of the domain to prove that there is no other moving object in the way. This is done using interpolation techniques.

On success the position of the robot (in the simulated state) is updated and all actions for the destination point are activated. Together with the simulated position the control knowledge is updated. This includes the total time the robot spent moving and the best time achieved reaching the destination for the current goal.

The same operator may be used for other goals. For example, the `GoAway` goal simply requires the robot to move to a position that is not occupied by other robots and will not intersect their paths of movement. In combination with a slight variation of the move operator, which ignores idle robots standing in the way, it results in the mutual behavior where robots move away from the path of another robot if it requests the right to pass by.

It was observed that the planning takes very little time for this small domain. The approximate time of path planning on an i686 3GHz machine is from 1ms to 10ms. This is a merit of the fact that the way point sub-domain is mainly static in its nature and all actions can be precomputed in advance. The operators can also take the advantage of the activation-change mechanism described above.

Foreign intentions. For coordination purposes, when a robot makes an intention to move somewhere, it communicates this intention to other robots. The trajectory of the movement, described by its points and exact time values, is sent to each robot so it can update its dynamic models of the domain. This information is stored in the `TimeModel` in the robot controller and used to estimate the positions of robots through the time. This valuable information guides the planning process of each robot so none of them intersects paths of the other.

Apart from the specific information sent by robots when they commit to a path, the dynamic model may be updated by the store. Failures[6] can also be used to update the time model. The store agent sends updates at defined time interval describing the situation at the storehouse in form of a snapshot. This information includes all robots and packets inclusive their positions, and all trucks coming in and going out.

[5]The code of `Move($to)` is provided in Appendix A.2.

[6]Communicated by other robots when they must drop a plan, because it cannot be pursued anymore.

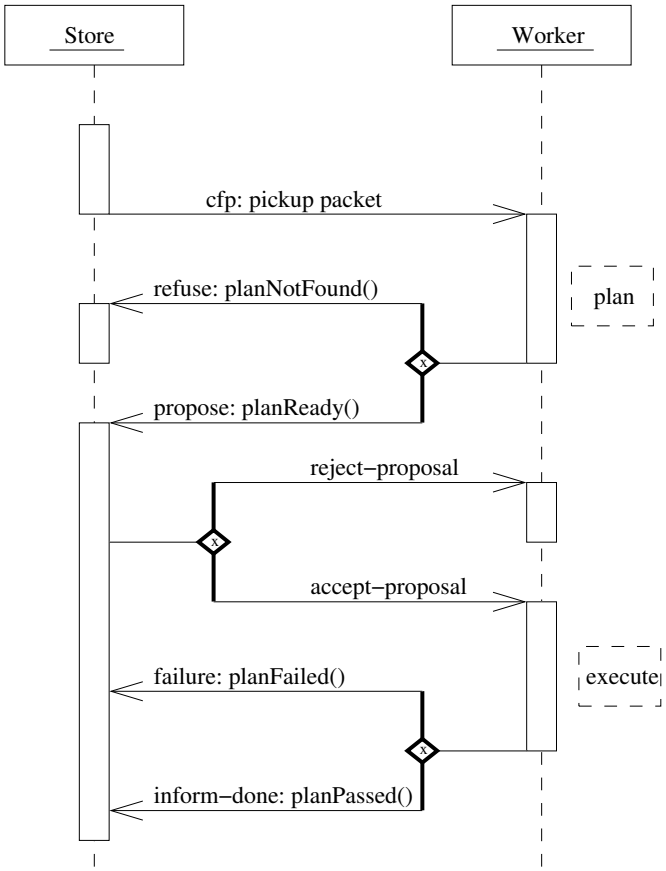

Figure 6.5: The Pickup Protocol used by Store and Workers to handle out plans for unloading a truck is based directly on the FIPA Contract Net Interaction Protocol.

Allocation of goals. The store agent simulates the trucks, which are handled by workers. It is also responsible for distributing goals like picking up a packet from the truck or delivering a packet of specified color to the truck. This is performed using a protocol based on the FIPA Contract Net Interaction Protocol depicted in Figure 6.5.

Whenever a truck arrives with packets to be unloaded, for example, the store agent sends in timely delayed intervals calls for proposal to worker agents for each packet. The `PickupPacketC` agent action sent by the store contains the name and position of a packet, as well as a deadline to be kept. It initiates the planner in each worker agent. If an agent has found a plan to pickup the packet, it sends a `PickuPacketP` proposal back to the store, or it sends a reject in the other case. The store chooses the plan promising to pickup the packet as early as possible and sends an *accept proposal* response to the winner. All other agents are quitted with an *reject proposal* message.

The agent responsible for the goal informs others about its intention and moves to the packet position, picks it up and places it anywhere in the store. At the time of planning and acting the agents do not accept any other goals or calls for proposals, as they can only hold one packet at the time.

6.3 Summary

Both domains presented in this chapter demonstrate a successful use of the planner concept implemented in the diploma thesis project. It gives comparable results in simple propositional domains like blocks-world and is capable of expressing planning knowledge using full programming language like JAVA™ in an object-oriented way. It showed remarkable performance in both domains of application.

The second example proves the usability of this concept for multi-agent domains including timing constraints and real valued attributes. The planner was successfully applied to aid the coordination among virtual robots in a simple two dimensional domain like the storehouse. This result also shows that the planner is a felicitous extension to the JADEX system and supplements it with reasonable operational capabilities.

Chapter 7

Conclusion

Planning – from the viewpoint of artificial intelligence – is one of the most important problem solving techniques. Classical planning was applied mostly to problem domains having the form of a propositional puzzle. The research field presents a wide landscape of planning techniques used to solve such problems in more or less efficient manner.

The concept of agency joins the views of artificial intelligence and computer science to envision reliable, efficient, flexible and autonomous software architectures capable to deal with information at semantic levels. The new research field proposes a wide range of systems ranging from simple reflex based agents to deliberative architectures claiming to be adequate – in respect to the Unified Theory of Cognition – to model human reasoning. One of the most successful architectures is based on the Procedural Reasoning System explained by the BDI model of the Theory of Practical Reasoning.

This thesis aimed towards merging the symbolic planning power of artificial intelligence planning techniques with the reactive and goal-oriented procedural reasoning capabilities offered by the BDI system of concern – JADEX. In respect to this main goal, the questions have been investigated, what planning techniques are adequate for the use in BDI systems in theory and practice, and how to define planning problems using JAVA™ language and process them efficiently with the help of a planner implemented in JAVA™.

7.1 Synopsis

A planner needs a representation of the planning problem, including the representation of states and changes, as well as goals. Different planning techniques can be applied to different domains. Chapter 2 presented goal stack, partial order, hierarchical and deductive planning techniques. Simple state-space based techniques have been mentioned together with the description of planning spaces. It was concluded that goal stack based techniques are not effective to cope with a general class of planning problems. Also partial order planners suffer performance losses due to the complexity of their representation and plan space explosions difficult to subdue. Deductive and transforming planners felt out of this project scope due to the lack

of efficient interpreters of declarative languages available for JAVA™ and the need to provide additional knowledge in form of frame axioms. The remaining techniques using hierarchical and state-space planning with domain specific control knowledge showed to be promising with respect to the application in BDI systems.

The results concerning planning concepts and methods have been verified in Chapter 3 on the basis of existing planning systems used in practice and introduced in different planning competitions. Particular emphasis was posed onto planners claiming generality and using *planning graph* techniques to achieve remarkable performance. A simpler and equally effective approach has been identified on the basis of state-space planners applying domain specific control knowledge. Another class of effective planners using domain specific knowledge are planners using hierarchical task networks formalism. The planners investigated use and work with declarative representation of planning problems like the PDDL. Because one of the aims of this thesis project was to provide a planner able to process JAVA™ language representation of goals, domains and operators, it has been concluded to use the simpler state-space based approach and design and implement such a planner.

The Theory of Practical Reasoning was mentioned in Chapter 4 together with the BDI model of agency based thereupon. Some theoretical and practical interpretations of this theory followed. BDI based agent architectures with and without a planner have been investigated. Several ways of merging the concept of AI planning with BDI systems have been identified. There are planner centered BDI architectures like RETSINA and there are composed systems like PROPICE-PLAN or CYPRESS. Other systems also describe ways to integrate a planner, but do not reveal details of such an implementation. The thesis project has chosen the composition based approach, mainly because the BDI system already existed.

With Chapter 5 a design of a planner has been shown that is capable to plan using an object-oriented language and representation of planning concepts. The planner utilizes domain control knowledge specified in operators and goals. It works on a meta-level representation of objects and their states that can be directly derived from agent beliefs. It is a reasonable extension to the JADEX BDI engine and aids agents in operational tasks that require symbolic manipulation and simulation of future states and changes.

Unfortunately, a direct relationship between AI planning concepts and BDI mental attitudes could not be established here. This is due to the highly advanced representation of goals and plans[1] used in JADEX, and due to the need to include domain specific control knowledge in operators as well as achievement metrics in goals.

Two example domains have been presented in the Chapter 6 that benefit from the planning component. Blocks-world is a standard domain used to evaluate planners and despite its simplicity, it is a challenge not to be underestimated. The planner performed quite well in this domain, crafting plans for problems including more than 500 blocks. The Loader Dock domain presents a multi-agent cooperative setting. The planner has been shown to help agents coordinate their activities. Also

[1] In relation to the simple symbolic representation used by PRS and PROPICE.

a distributed goal solving scenario has been investigated.

Shortcomings. The control knowledge used by the operators has the negative side effect that operators cannot be applied to a problem domain orthogonally. It was observed, that adding new operators requires modifying control knowledge of other ones.

Because the domain description is created dynamically, the predicates used by operators need access to attribute indexes as registered by the operator. This reduces the reusability of predicates, because of the tight coupling to the operators.

The activation-change mechanism introduced in the planner is a weak form of forward chaining networks. It does not suffice for all purposes, because the activation is only bound to an object or variable and cannot be bound to an expression. The activation ceases in the following steps and must be renewed in some way, causing further dependencies among operators.

On the other hand, putting a full blown activation mechanism using a forward chaining approach with a generic way of representing expressions as explicit sets of objects, would not be feasible because of space problems. There is a question emerging here in respect to the transitional frame problem and to the representation of states using such an approach.

7.2 Outlook

The planner provides some capability for hierarchical plan decomposition and, thus, may be used to perform a simple kind of hierarchical planning. There are many other concepts that have been investigated with hierarchical planners. There was no simple solution – known to the author – that would offer a practically usable hierarchical planner working with JAVA™ representation. This thesis work had no time to create a complex full blown hierarchical planner using partially lifted representations and unordered decomposition. It is assumed here that it would be more complex – compared to the simple (one goal at a time) approach taken here – to apply heuristics and control knowledge to HTN.

The design of JADEX has been guided by more practical concerns than a provision of a frame work for testing artificial intelligence concepts. A full synthesis system of the BDI agency model and the hierarchical planning approach – in the sense of the planning component from RETSINA – that is fully based on an object-oriented representation and augmented with control knowledge and heuristic methods is certainly also as interesting to investigate as the composition approach taken here.

Systems based on the Procedural Reasoning System have a neat property that goals can always be given up, whenever all plans applicable for this goal fail. The Theory of Practical Reasoning foresees the process of giving up intentions only if new conflicting intensions have been adopted in the process of deliberation. When the architecture is extended with a symbolic AI planner, the number of plans is no more bounded. A simple BDI agent provided with unlimited number of plans would

do Sisyphus work always assuming the goal can be achieved in some new way. The notion of goal failure must be reconsidered in such a case.

The implementation of the planner, example domains and additional utilities contains about 24000 lines of code. It is surely a prototypic work making about 10% of JADEX, so there is certainly room for improvements in respect to this size. Especially a common parser and term representation of JAVA™ expressions in the main project would contribute to this concern.

MAS Planning. The thesis includes an evaluation of a problem solved in a distributed way. In respect to this, some questions arise regarding mutual representation of goals, plans and intentions. How to formalize and communicate mutual attitudes? Also intentions are entities with high degree of commitment. BDI models do not regard options and hypotheses as their primary concepts. The aspect of sharing such virtual notions of mind among agents would certainly help solving distributed planning problems.

Search. It is interesting, how other relatively fast and successful search strategies could be adapted for the problem space. The questions is, how to define knowledge based control and heuristic rules for the backward search regression algorithm. The backward search would have a rudimentary presence of online planning capability to its advantage and much space requirements for domains with large state models and with many attributes.

The global search keeps the most promising partial solutions in its limited agenda. The choice of solutions on the agenda is determined by the heuristics of the domain; this biases it towards a fairly homogeneous collection. There should be means that allow to investigate interference between these partial solutions. The interference could be used to spread the collection to many points in the search space and keep only representatives on the agenda. To use such feature one should define similarity metrics on partial solutions.

Tools. There is a requirement for tools supporting the development of a planning domain. A domain designer would certainly appreciate a tool with graphical input and JAVA™ code output. State descriptions could be generated that are easily copied and manipulated without side effects. This would replace the troublesome generic representation of the `IndexedState` and spare the operator programmer from registering attributes. The only question to be answered here is, how to solve the problem of technical reference – also known as direct object association – among JAVA™ objects in a cloned state. It is similar to the problem of serializing objects with cyclic dependencies. The thesis sidesteps this problem with help of virtual references and hash maps on the meta-level.

The development of effective control knowledge is time consuming and should be supported by a tool debugging the planning process. Such a tool would have the capability to view the whole hypothesis space created by the planner, so ineffective branches of the search space may be identified in its structure.

Bibliography

Allen, J. (1991), Planning as temporal reasoning, *in* '2nd Principles of Knowledge Representation and Reasoning', Morgan Kaufmann.

AOS (2004), *JACK Intelligent Agents™ Agent Manual*, 4.1 edn, Agent Oriented Software Pty. Ltd.

Arentoft, M. M., Fuchs, J. J., Parrod, Y., Gasquet, A., Stader, J., Stokes, I. & Vadon, H. (1992), 'OPTIMUM-AIV: a planning and scheduling system for spacecraft AIV', *Future Gener. Comput. Syst.* **7**(4), 403–412.

Bacchus, F. & Ady, M. (1999), *Precondition control*, University Of Toronto. *http://www.cs.toronto.edu/∼ fbacchus/on-line.html

Bacchus, F. & Kabanza, F. (2000), 'Using temporal logics to express search control knowledge for planning', *Artificial Intelligence* **116**(1-2), 123–191.

Bellifemine, F., Poggi, A. & Rimassa, G. (1999), JADE - a FIPA - compliant agent framework, *in* '4th International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-99)', The Practical Application Company Ltd., London, UK.

Blum, A. L. & Furst, M. L. (1997), 'Fast planning through planning graph analysis', *Artificial Intelligence* **90**, 281–300.

Bonet, B. & Geffner, H. (2001), 'Heuristic search planner 2.0', *The AI Magazine* **22**(1), 77–80.

Bonet, B., Loerincs, G. & Geffner, H. (1997), A robust and fast action selection mechanism for planning., *in* 'Proceedings of the AAAI-97', MIT Press, pp. 714–719.

Bratman, M. E. (1987), *Intention, Plans, and Practical Reason*, Harvard University Press, Cambrige, MA.

Bratman, M. E., Israel, D. J. & Pollack, M. E. (1988), 'Plans and resource-bounded practical reasoning', *Computational Intelligence* **4**, 349–355.

Braubach, L., Pokahr, A., Krempels, K.-H. & Lamersdorf, W. (2004), Deployment of distributed multi-agent systems, *in* M.-P. Gleizes, A. Omicini & F. Zambonelli, eds, 'Fifth International Workshop on Engineering Societies in the Agents World'.

Braubach, L., Pokahr, A. & Lamersdorf, W. (2004), Jadex: A short overview, *in* 'Main Conference Net.ObjectDays 2004', pp. 195–207.

Braubach, L., Pokahr, A., Moldt, D. & Lamersdorf, W. (2004), Goal representation for BDI agent systems, *in* 'The Second International Workshop on Programming Multi Agent Systems', pp. 9–20.

Busetta, P., Howden, N., Rönnquist, R. & Hodgson, A. (2000), Structuring BDI agents in functional clusters, *in* N. Jennings & Y. Lespérance, eds, 'Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL'99), LNCS 1757', Springer-Verlag, pp. 277–289.

Busetta, P., Ronnquist, R., Hodgson, A. & Lucas, A. (1999), 'JACK intelligent agent - components for intelligent agents in Java'.

Bylander, T. (1992), Complexity results for serial decomposability., *in* 'Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)', AAAI Press, Syn Jose, pp. 729–734.

Bylander, T. (1994), 'The computational complexity of propositional STRIPS planning', *Artificial Intelligence* **69**, 165–204.

Carbonell, J. G., Knoblock, C. A. & Minton, S. (1991), Prodigy: An integrated architecture for planning and learning., *in* K. VanLehn, ed., 'Architectures for Intelligence', Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 241–278.

Cohen, P. R. & Levesque, H. J. (1990), Persistence, intention, and commitment, *in* P. R. Cohen, J. Morgan & M. E. Pollack, eds, 'Intentions in Communication', MIT Press, Cambridge, MA, pp. 33–69.

de Silva, L. & Padgham, L. (2004), A comparison of BDI based real-time reasoning and HTN based planning., *in* 'AI 2004: Advances in Artificial Intelligence, 17th Australian Joint Conference on Artificial Intelligence', Springer, Cairns, Australia, pp. 1167–1173.

Despouys, O. & Ingrand, F. F. (1999), Propice-plan: Toward a unified framework for planning and execution., *in* S. Biundo & M. Fox, eds, 'Recent Advances in AI Planning, 5th European Conference on Planning, ECP'99', Springer, Durham, UK, pp. 278–293.

d'Inverno, M., Kinny, D., Luck, M. & Wooldridge, M. (1997), A formal specification of dMARS, *in* 'Agent Theories, Architectures, and Languages', pp. 155–176.

Doherty, P., Gustafsson, J., Karlsson, L. & Kvarnström, J. (1998), 'TAL: Temporal action logics language specification and tutorial', *Electronic Transactions on Artificial Intelligence* **2**(3-4), 273–306.
*http://www.ep.liu.se/ej/etai/1998/009/

Doherty, P. & Kvarnström, J. (2001), 'TALplanner: A temporal logic-based planner', *The AI Magazine* **22**(1), 95–102.

Duvigneau, M., Moldt, D. & Rölke, H. (2003), Concurrent Architecture for a Multi-agent Platform, *in* F. Giunchiglia, J. Odell & G. Weiß, eds, 'Agent-Oriented Software Engineering III. (AOSE) 2002', Vol. 2585 of *LNCS*, Springer, Berlin Heidelberg New York.

Edelkamp, S. & Helmert, M. (2000), 'On the implementation of MIPS', Paper presented at the Fifth International Conference on Artificial Intelligence Planning and Scheduling, Workshop on Model-Theoretic Approaches to Planning. Breckenridge, Colorado, 14 April.

Edelkamp, S. & Hoffmann, J. (2004), PDDL2.2: The language for classical part of the 4th international planning competition, Technical report, 4th International Planning Competition.
*http://ls5-www.cs.uni-dortmund.de/∼edelkamp/ipc-4/DOCS/pddl2.2.ps.gz

Edelkamp, S., Hoffmann, J., Littman, M. & Younes, H. (2004), 'The 4th international planning competition 2004 (IPC-2004)'. Hosted at the International Conference on Automated Planning and Scheduling 2004 (ICAPS-2004).
*http://ls5-www.cs.uni-dortmund.de/∼edelkamp/ipc-4/

Erol, K., Hendler, J. A. & Nau, D. S. (1994), UMCP: A sound and complete procedure for hierarchical task-network planning, *in* 'Artificial Intelligence Planning Systems', pp. 249–254.

Fikes, R. E. & Nilsson, N. J. (1971), 'STRIPS: a new approach to the application of theorem proving to problem solving', *Artificial Intelligence* **2**(3–4), 189–208.

Firby, R. J. (1989), Adaptive execution in complex dynamic worlds, Ph.D. Thesis YALEU/CSD/RR #672, Yale University.

Fischer, K., Müller, J. P. & Pischel, M. (1994), Unifying control in a layered agent architecture, Technical Report TM-94-05, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Kaiserslautern, DE.

Fox, M. & Long, D. (2001), PDDL 2.1: An extension to PDDL for expressing temporal planning domains, Technical report, Department of Computer Science, University of Durham, UK.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Georgeff, M. & Lansky, A. (1986), 'Procedural knowledge', *Proceedings of the IEEE (Special Issue on Knowledge Representation)* **74**, 1383–1398.

Georgeff, M. P. & Ingrand, F. F. (1990), Real-time reasoning: The monitoring and control of spacecraft systems, *in* 'Proceedings of the Sixth Conference on Artificial Intelligence Applications CAIA-90 (Volume I: Papers)', Santa Barbara, CA, pp. 198–204.

Georgeff, M. P. & Lansky, A. L. (1987), Reactive reasoning and planning: An experiment with a mobile., *in* 'Proceedings of the sixth National Conference on Artificial Intelligence (AAAI-87)', Seattle, Washington, pp. 677–682.

Georgeff, M. P., Pell, B., Pollack, M., Tambe, M. & Wooldrige, M. (1998), The belief-desire-intention model of agency, *in* 'Intelligent Agents, 5th International Workshop, ATAL'98', Springer, Paris, pp. 1–10.

Gerevini, A. & Serina, I. (2002), LPG: a planner based on local search for planning graphs, *in* 'Proceedings of the Sixth Internationl Conference on AI Planning and Scheduling (AIPS'02)', AAAI Press, pp. 13–22.

Ghallab, M., Nau, D. & P.Traverso (2004), *Automated Planning: Theory and Practice*, Morgan Kaufmann Publishers.

Graham, J. R. & Decker, K. S. (2000), Towards a distributed, environment-centered agent framework, *in* 'Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL 99), LNCS 1757', Springer-Verlag, pp. 290–304.

Graham, J. R., Decker, K. S. & Mersic, M. (2003), 'Decaf - a flexible multi agent system architecture', *Autonomous Agents and Multi-Agent Systems* **7**(1-2), 7–27.

Green, C. (1969), Application of theorem proving to problem solving, *in* 'Proc. of the 1st IJCAI', Washington, DC, pp. 219–239.

Hayes-Roth, B. & Hayes-Roth, F. (1979), 'A cognitive model of planning', *Cognitive Science* **3**, 275–310.

Henckel, J. D. (1993), Theorem prover with natural language interface, Master's thesis, University of Minnesota.
*http://www.geocities.com/Paris/6502/unif.html

Hoffmann, J. & Nebel, B. (2001), 'The FF Planning System: Fast plan generation through heuristic search', *Journal of Artificial Inteligence Research* **14**, 253–302.

Huber, M. (1999), JAM: A BDI-theoretic mobile agent architecture, *in* O. Etzioni, J. P. Müller & J. Bradshaw, eds, 'Proceedings of the Third Annual Conference on Autonomous Agents', ACM Press, pp. 236–243.

Ilghami, O. & Nau, D. S. (2003), A general approach to synthesize problem-specific planners, Technical Report CS-TR-4597, Department of Computer Science, University of Maryland.

Jennings, N. R. (1999), Agent-based computing: promise and perils., *in* 'Proceedings of the 16th International Joint Conference on Artificail Intelligence (IJCAI-99)', Stockholm, Sweden, pp. 1429–1436.

Kautz, H. & Selman, B. (1992), Planning as satisfiability, *in* B. Neumann, ed., 'Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)', Wiley, Vienna, Austria, pp. 359–363.

Kautz, H. & Selman, B. (1999), Unifying SAT-based and graph-based planning., *in* 'Proceedings of the 16th International Joint Conference on Artificial Inteligence (IJCAI-99)', Morgan Kaufmann, Stockholm, Sweden, pp. 318–325.

Koehler, J., Nebel, B. & Hoffmann, J. (1997), Extending planning graphs to an ADL subset, Technical Report 88, Institute for Computer Science – Albert Ludwigs University, Freiburg, Germany.
*http://www.informatik.uni-freiburg.de/∼koehler/ipp.html

Kvarnström, J. & Magnusson, M. (2003), 'TALplanner in IPC-2002: Extensions and control rules', *Journal of Artificial Intelligence Research (JAIR)* **20**, 343–377.

Laird, J., Newell, A. & Rosenbloom, P. (1987), 'SOAR: An architecture for general intelligence.', *Artificial Intelligence* **33**(1), 1–67.

Lin, F. (2001), 'A planner called R', *The AI Magazine* **22**(1), 73–76.

Long, D. & Fox, M. (2002), 'The 3rd international planning competition 2002 (IPC-2002)', http://planning.cis.strath.ac.uk/competition/. Hosted at the Artificial Intelligence Planning and Scheduling Conference 2002 (AIPS-2002).

McCarthy, J. (1963), 'Situations, actions and causal laws', Stanford Artificial Intelligence Project: Memo 2.

McCarthy, J. & Hayes, P. J. (1969), Some philosophical problems from the standpoint of artificial intelligence, *in* 'Machine Intelligence 4', Edinburgh University Press, Edinburgh, Scotland, pp. 463–502.

McDermott, D. (1991), 'Regression planning.', *International Journal of Intelligent Systems* **6**, 357–416.

McDermott, D. & AIPS'98 Commitee (1998), 'PDDL - the planning domain definition language', http://www.cs.yale.edu/homes/dvm.

Meneguzzi, F. R., Zorzo, A. F. & da Costa Móra, M. (2004), Propositional planning in BDI agents, *in* 'Proceedings of the 2004 ACM symposium on Applied computing', ACM Press, pp. 58–63.

Móra, M. C., Lopes, J. G., Viccari, R. M. & Coelho, H. (1999), BDI models and systems: Reducing the gap., *in* 'Proceedings of the 5th International Workshop on Intelligent Agents', Springer.

Myers, K. L. (1997), *User Guide for the Procedural Reasoning System*, Artificial Intelligence Center, SRI International, Menlo Park, CA.

Myers, K. L. & Wilkins, D. E. (1997), *The Act Formalism*, 2.2 edn, SRI International Artificial Intelligence Center, Menlo Park, CA.

Nau, D. S., Au, T. C., Ilghami, O., Kuter, U., Murdock, W., Wu, D. & Yaman, F. (2003), 'SHOP2: An HTN planning system', *Journal of Artificial Intelligence Research* **20**, 379–404.

Nau, D. S., Cao, Y., Lotem, A. & Muñoz-Avila, H. (1999), SHOP: Simple hierarchical ordered planner, *in* 'Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence', Morgan Kaufmann Publishers Inc., pp. 968–975.

Nau, D. et al. (2004), Applications of SHOP and SHOP2, Technical Report CS-TR-4604, UMIACS-TR-2004-46, University of Maryland.

Newell, A. & Simon, H. A. (1963), GPS, a program that simulates human thought, *in* E. A. Feigenbaum & J. Feldman, eds, 'Computer and Thought', 279–293, McGraw-Hill, New York.

Nguyen, X. & Kambhampati, S. (2001), Reviving partial order planning., *in* B. Nebel, ed., 'Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence IJCAI 2001', Morgan Kaufmann, Seattle, Washington, USA, pp. 459–466.

Nguyen, X., Kambhampati, S. & Nigenda, R. S. (2002), 'Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search', *Artif. Intell.* **135**(1-2), 73–124.

Paolucci, M., Shehory, O., Sycara, K. P., Kalp, D. & Pannu, A. (2000), A planning component for retsina agents, *in* 'ATAL '99: 6th International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL),', Springer-Verlag, pp. 147–161.

Pednault, E. (1989), ADL: Exploring the middle ground between STRIPS and the situation calculus, *in* '1st Int. Conf. on Principles of Knowledge Representation and Reasoning', pp. 324–332.

Penberthy, J. S. & Weld, D. S. (1992), UCPOP: A sound, complete, partial order planner for ADL, *in* B. Nebel, C. Rich & W. Swartout, eds, 'KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference', Morgan Kaufmann, San Mateo, California, pp. 103–114.

Pokahr, A., Braubach, L. & Lamersdorf, W. (2003), 'Jadex: Implementing a BDI-infrastructure for JADE agents', *EXP - In Search of Innovation (Special Issue on JADE)* **3**, 101–117.

Refanidis, I. & Vlahavas, I. P. (2001), The GRT planner: New results, *in* 'Proceedings of the Workshop on Local Search for Planning and Scheduling-Revised Papers', Springer-Verlag, pp. 120–138.

Russell, S. J. & Norvig, P. (2003), *Artificial Intelligence: A Modern Approach (2nd Edition)*, 2 edn, Prentice Hall.

Sacerdoti, E. D. (1974), 'Planning in a hierarchy of abstraction spaces', *Artificial Intelligence* **5**, 115–135.

Sacerdoti, E. D. (1975), The nonlinear nature of plans., *in* 'Proc. of the 4th IJCAI', Tbilisi, Georgia, USSR, pp. 206–214.

Seegert, V. (2004), Untersuchung von planerkonzepten für mulan-agenten, Master's thesis, University of Hamburg.

Shafer, G. (1976), *A Mathematical Theory of Evidence*, Princeton University Press, Princeton, NJ.

Shut, M. & Wooldridge, M. (2001), 'The control of reasoning in resource-bounded agents.', *The Knowledge Engineering Review* **16**(3).

Sycara, K., Paolucci, M., Velsen, M. V. & Giampapa, J. (2003), 'The RETSINA MAS infrastructure', *Autonomous Agents and Multi-Agent Systems* **7**(1-2), 29–48.

Tate, A., Trabble, B. & Dalton, J. (1996), 'O-Plan: A knowledge-based planner and its application to logistic', *Advance Planning Technology* pp. 213– 239.

Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., Fink, E. & Blythe, J. (1995), 'Integrating planning and learning', *Journal of Experimental and Theoretical Artificial Intelligence* **7**(1).

Wah, B. W. & Chen, Y. X. (2003), Partitioning of temporal planning problems in mixed space using the theory of extended saddle points, *in* 'Proceedings of IEEE Int'l Conf. on Tools with Artificial Intelligence', IEEE Computer Society, Sacramento, California, USA, pp. 266–273.

Weld, D. S., Anderson, C. R. & Smith, D. E. (1998), Extending graphplan to handle uncertainty and sensing actions, *in* 'Proceedings of AAAI'98'.

Wilensky, R. (1983), *Planning and Understanding. A Computational Approach to Human Reasoning*, Addison-Wesley, Reading, MA.

Wilkins, D. E. (1988), *Practical Planning: Extending the Classical AI Planning Paradigm*, Morgan Kaufmann, San Mateo, CA.

Wilkins, D. E. (1999), Using the SIPE-2 planning system: A manual for version 6.1, Technical report, SRI International Artificial Intelligence Center, Menlo Park, CA.

Wilkins, D. E., Myers, K. L. & Wesley, L. P. (1994), Cypress: Planning and reacting under uncertainity, *in* M. H. Burstein, ed., 'ARPA/Rome Laboratory Planning and Scheduling Initiative Workshop Proceedings', Morgan Kaufmann Publishers Inc., San Mateo, CA, chapter 111–120.

Wooldridge, M. (1997), 'Agent-based software engineering', *IEE Proceedings Software Engineering* **144**(1), 26–37.

Appendix A

Code Samples

A.1 Unification Algorithm

```
BOOL unify(n1, v1, n2, v2)
   IF n1 and n2 are variables
      IF n1 and n2 are unbound
         Bind the variable with the shorter
         chain to the variable with the longer chain
         by appending the latter to the first.
         RETURN TRUE
      ELSE
      IF n1 is unbound in v1
         Bind n1 to n2 by appending the latter to the first.
         RETURN TRUE
      ELSE
      IF n2 is unbound in v2
         Bind n2 to n1 by appending the latter to the first.
         RETURN TRUE
      ELSE
      IF n1 and n2 are bound to an Object
         RETURN object of n1 equals object of n2
      ELSE
      IF n1 is bound to an Object
         RETURN unify(n1, v1, the term instance bound to n2)
      ELSE
      IF n2 is bound to an Object
         RETURN unify(the term instance bound to n1, n2, v2)
      ELSE // both are bound to term instances
         RETURN unify(the term instance of n1, the term instance of n2)
   ELSE
   IF n1 is a variable
      IF n1 is unbound
         Bind it to the term instance defined by n2 and v2
         RETURN TRUE
      ELSE
      IF n1 is bound to an object
         IF n2 is a Value node
            RETURN the object of n1 equals to the value of n2
         ELSE
```

```
            RETURN FALSE
      ELSE // n1 is bound to a term instance
         RETURN unify(the term instance of n1, n2, v2)
   ELSE
   IF n2 is a variable
      ... // symmetric to the case above
   ELSE
      // both are not variables
      IF type of n1 != type of n2
         RETURN FALSE
      ELSE
      IF n1 and n2 are Symbols
         RETURN n1 == n2
      ELSE
      IF the number of children is equal for n1 and n2
         // unify child nodes
         FOR_EACH child node c1 of n1
            IF NOT unify(c1, v1, corresponding child node of n2, v2)
               RETURN FALSE
         END_FOR
         RETURN TRUE
      ELSE
         RETURN FALSE
END // of unify
```

A.2 Worker Move Operator

```
public class Move implements Operator {
  /** used to store total time of movement */
  public static final Var MOVE_TIME=new Var("move_time");

  /** the destination */
  protected final Point to;

  /**  the robot controller */
  protected RobotController ctrl;

  /** Constructor: <code>Move</code>.
   * @param to is the destination of this move operator
   */
  public Move(Point to) {
    if (to==null || to.hasType(Point.TYPE_OBSTACLE)) {
     throw new IllegalArgumentException("Illegal point");
    }
    this.to   = to;
  }

  /** The setup method used to initialize the operator.
   * @param dd the domain description used by the setup
   */
  public void setup(Description dd){
    ctrl=(RobotController)dd.get(Robot.CTRL);
```

```
  Point[] ps=ctrl.getHouseModel()
                 .getNeighbours(to.getX(),
                                to.getY(),
                                0.1,
                                FloorPlan.GRID_DISTANCE);
  int i=ps.length;
  if (i==0) {
    throw new IllegalArgumentException("Cannot move here");
  }
  while(i-->0) {
    dd.activateOnChange(this, ps[i]);
  }
}

/** The application method including preconditions,
 * control knowledge and effects.
 * @param st the step to apply the changes to
 */
public boolean applyTo(Step st)  {
  if (!(st.getGoal() instanceof GoGoal)) return false;

  GoGoal go=(GoGoal)st.getGoal();

  if (!validPath(st)) return false;

  Point  cur=(Point)st.get(Robot.POS);
  if (ctrl.getHouseModel()
          .hasObstacle(cur.getX(),
                       cur.getY(),
                        to.getX(),
                        to.getY())) {
     return false;
  }

  // compute the distance
  double dx=to.getX()-cur.getX();
  double dy=to.getY()-cur.getY();
  double ds=Math.sqrt(dx*dx+dy*dy);
  if (ds<5.0 || ds>FloorPlan.GRID_DISTANCE) return false;

  // compute the heading adjustment
  double cd=((Double)st.get(Robot.DIR)).doubleValue();
  double nd=Math.atan2(dx, -dy);
  double da=Robot.norm(nd-cd);

  da*=1.2; // overdrive

  if (da>Math.PI*.5 || da<-Math.PI*.5)  {
    // move backwards
    da = Robot.norm(nd-cd+Math.PI)*1.2;
  }

  nd = Robot.norm(da+cd);
  if (da<0.0) da=-da; // absolute value
```

```
  // compute the time this action will be finished
  double move_time=ds*Robot.LOAD_FACTOR/Robot.SPEED +
                   da/Robot.TURN_SPEED+Robot.BUFFER;
  double fin_time=st.getTime()+move_time;

  double total_mtime=st.getDouble(MOVE_TIME);

  // access control knowledge in the goal
  Double tD=(Double)go.p2t.get(to);
  if (tD!=null && total_mtime>tD.doubleValue()) return false;

  if (!colision(cur, st.getTime()-Robot.BUFFER,
                     fin_time+Robot.BUFFER))  {
    // update control knowledge
    total_mtime+=move_time;
    go.p2t.put(to, new Double(total_mtime));
    st.set(MOVE_TIME, new Double(total_mtime));

    // apply changes
    st.set(Robot.DIR, new Double(nd));
    st.set(Robot.POS, to);

    st.setTime(fin_time);
    st.activateActionsFor(to);
    return true;
  }

  return false;
}

/** Tests for collision using the dynamic domain model.
 * @param cur
 * @param t1
 * @param t2
 * @return true if there is a collision on the way
 */
protected boolean colision(Point cur, double t1, double t2) {
  return
     ctrl.getTimeModel().conflictsIdle(cur, to, t1, t2) ||
     ctrl.getTimeModel().hasObstacle(cur, to, t1, t2);
}

/** Tests if the plan prefix is a valid one.
 * @param st
 * @return false if there was a position equal to this one.
 */
protected boolean validPath(Step st) {
  while(st!=null)  {
    Point prev=(Point)st.get(Robot.POS);
    if (to.distance(prev)<5.0) return false;
    st = st.getPrevious();
  }
  return true;
```

```
  }

  /** Performs this action in the real environment.
   * @return true if the movement succeeded
   */
  public boolean perform(Object arg, double deadline) {
    Plan plan=(Plan)arg;
    return ctrl.performMove(plan, to, deadline);
  }
}
```