Planning and Choosing: Augmenting HTN-Based Agents with Mental Attitudes*

Gerhard Wickler, Stephen Potter, Austin Tate
AIAI, School of Informatics, University of Edinburgh
g.wickler|s.potter|a.tate@ed.ac.uk

Michal Pěchouček, Eduard Semsch
Agent Technology Group, Department of Cybernetics, Czech Technical University
pechouc|semsch@labe.felk.cvut.cz

Abstract

This paper describes a new agent framework that fuses an HTN planner, through its underlying conceptual model, with the mental attitudes of the BDI agent architecture, thus exploiting the strengths of each. On the one hand, the practical and proven ability to reason about actions that is the strength of HTN planning fleshes out the option generation function in the inference loop of the BDI model; on the other hand, the mental attitudes make explicit the knowledge that plays an essential role in plan selection, an important aspect that is not considered in the traditional formulation of the planning problem. The result is a coherent framework that allows for the design and implementation of activity-centric rational agents.

1. Introduction

I-X [11] is an agent framework based on the HTN (Hierarchical Task Networks) planning paradigm [7, 10]. HTN planners have been successfully used in a number of real-world applications [4] and agents based on this framework inherit the advantages of this approach. I-X agents have an internal state which is essentially a plan. The conceptual model used for the representation of the agent's internal state as a plan is called <I-N-C-A> [12] and a formalization of this model will be described here.

However, lacking in the <I-N-C-A> representation are the mental attitudes that define the BDI agent architecture, namely an agent's *beliefs*, *desires* and *intentions* [6, 15]. This paper presents a refinement of the <I-N-C-A> model, incorporating the mental attitudes that constitute the BDI model into <I-N-C-A>. In this elaborated model, *beliefs* and *intentions* can be represented quite naturally and explicitly as constraints and activities respectively in <I-N-C-A>. *Desires* are contained implicitly in the decision-making knowledge of the agent. The result is an agent framework that combines the strengths of both approaches: it fuses the option generation capability of a powerful HTN planner and its underlying representation with the mental attitudes of the BDI model that are utilized for choosing the best possible of the generated options as the course of action to be followed.

A similar approach is described in [8], where the CAN agent programming language has been extended to include behaviour produced by an HTN planner. In contrast, the starting point of the research reported here is a planning formalism, which, by the incorporation of BDI mental attitudes, results in a general activity-centric agent model, thereby providing a complementary account of the relationship between mental state and future action.

The remainder of this paper is organized as follows. A brief summary of the well-known BDI model is followed by a formalization of the <I-N-C-A> model and its main components. We then describe how <I-N-C-A> has been augmented using the mental attitudes from the BDI model. This is followed by an operational semantics of the resulting model that can be seen as the basis for an agent interpreter that renders the knowledge-level specification executable.

2. The BDI Agent Model

The BDI model is one of an agent that fulfills three key qualities: autonomy, reactivity and intentionality [16]. The BDI model does not explicitly support the social properties of an agent: the ability to communicate, cooperate and reason about other agents in a multi-agent community.

*Sponsored in part by the European Research Office of the US Army under grant number N62558-06-P-0353 and in part by Czech Ministry of Education grant 6840770038. The authors' organizations and research sponsors are authorized to reproduce and distribute reprints and on-line copies for their purposes notwithstanding any copyright annotation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of other parties.

A BDI agent is distinguished by the organization of its knowledge, which governs its behaviour, into three distinct knowledge structures based on the mental state modalities: *beliefs*, *desires* and *intentions* [6]. Beliefs represent the agent's current knowledge about itself and its environment, desires represent its longer term goals and objectives of its behaviour, and intentions represent the agent's local decisions about the actions it intends to perform.

A BDI agent executes its behaviour by manipulating its internal knowledge and data according to the standard BDI inference loop:

function BDI-inference-loop(Bel, Des, Int)
  while true **do**
    $p \leftarrow$ getNextPercept()
    $Bel \leftarrow$ beliefRevision(Bel, p)
    $Int \leftarrow$ generateOptions(Bel, Des)
    $Int \leftarrow$ filterIntentions(Bel, Des, Int)
    $plan \leftarrow$ generatePlan(Bel, Int)
    execute($plan$)

A BDI agent first processes the next external percept (such as an incoming message). As a result the agent revises its beliefs. Then, based on the new beliefs, it generates several different intentional options out of which one is adopted by means of the *filterIntention* function. Next, the *generatePlan* function elaborates a plan for the adopted intentions that is then executed. In applications of the BDI model this planning process is often based on pattern-matching against a library of predefined plans.

3. The <I-N-C-A> Model

<I-N-C-A> is a generic model for synthesis tasks [12]. While its level of abstraction makes it possible to apply the generic model to a wide variety of tasks, it assumes a more specific meaning in the I-X framework where an <I-N-C-A> object is synonymous with a plan—the course of action the I-X agent intends to follow—and the planning task can be thought of as one of synthesizing an appropriate object. A plan can be partial in the sense that it is not (yet) an actionable solution to the problem in question. The planning agent must refine a partial plan into a solution plan.

We can formally define an <I-N-C-A> object in I-X as a 4-tuple (I, N, C, A) consisting of: a set of *issues* I; a set of *activity nodes* N; a set of *constraints* C; and a set of *annotations* A.

3.1. Issues

Informally, issues with the current plan can be thought of as indicating flaws in the plan or opportunities that the planner might exploit. An <I-N-C-A> object is considered to be a solution to a planning problem only if the set of issues is empty, that is, all flaws have been rectified, and all opportunities considered.

More formally, I is the set of unresolved issues in the current plan, i.e., in this <I-N-C-A> object. An issue is represented by a syntactic expression of the form $l : M(O_1, \ldots, O_n)$, where:

- l is a unique label for this issue,
- M is a symbol denoting a primitive plan modification activity, and
- $O_1, \ldots, O_n$ are plan-space objects, i.e. issues, nodes, constraints or annotations. The number of objects, n, and the interpretation of each object in the context of the issue, will depend on the particular primitive plan modification activity represented by this issue.

Issues can be seen as primitive meta-level activities, i.e. things that need to be done to the plan before it becomes a solution to a given planning problem. This approach is inherited from O-Plan [2, 13] and is also seen in planners such as OPIS [9]. The most commonly found primitive meta-level activities carried out by planners, but usually only implicit in their underlying implementation or internal plan representation, are:

- Achieving a goal (in classical planning): Let p be a world-state proposition and τ be a time point, then the primitive meta-level activity of achieving p at τ can be represented as the issue:
$$l : \texttt{achieve}(p, \tau)$$
- Accomplishing a complex activity (in HTN planning): Let $a \in N$ be a complex activity. Then the primitive meta-level activity of accomplishing a can be represented as the issue:
$$l : \texttt{refine}(a)$$

Here, `achieve` and `refine` are examples of symbols denoting primitive plan modification activities. Note that these symbols are not domain specific but specific to the planning process by which these types of issue are handled.

3.2. Nodes

N is the set of activities (nodes) to be performed in the current plan, i.e., in this <I-N-C-A> object. An activity is a syntactic expression of the form $l : \alpha(o_1, \ldots, o_n)@[\tau_b, \tau_e]$, where:

- l is a unique label for this activity,
- α is a symbol denoting an activity name,

- $o_1, \ldots, o_n$ are object-level terms, i.e. they are either constant symbols describing objects in the domain, or they are as yet uninstantiated variables standing for such objects, and
- τ_b and τ_e are time points representing the beginning and the end of the activity.

Time points could be seen as another pair of parameters o_{n+1}, o_{n+2} of an activity but are given a special place in the syntax here due to the importance of time in planning. In the context of I-X, nodes represent the object-level activities in the plan, i.e., things that need to be performed by some agent to execute the plan. Activities can be of two types from the perspective of the planner:

- Primitive activities: primitive activities can be carried out directly by an agent executing the plan. For example, in a search and rescue domain, the primitive activity of flying the aircraft `ac1` from location `loc1` to location `loc2` may be represented as:
$$l : \texttt{fly}(\texttt{ac1},\texttt{loc1},\texttt{loc2})@[t_1, t_2]$$
- Complex activities: complex activities cannot be accomplished directly by the agent executing the plan but need to be refined into primitive activities. For example, the complex activity of rescuing an isolated person ip may be represented as:
$$l : \texttt{rescue}(\texttt{ip})@[t_1, t_2]$$

In this example, `fly` is a primitive activity symbol and `rescue` is a complex activity symbol in some domain. Activity symbols have to be domain specific. It follows that there has to be an activity schema defined for the domain that has the name `fly` and describes when this activity schema is applicable and how it will change the world when applied, and there has to be a refinement defined in the domain that accomplishes a complex activity with the name `rescue` and describes how exactly it can be accomplished.

Note that the set N of activities in the plan may contain both complex activities and the primitive activities that have been chosen to implement them.

3.3. Constraints

C is the set of constraints that must be satisfied by the current plan (<I-N-C-A> object). A constraint is a syntactic expression of the form $l : c(v_1, \ldots, v_n)$, where:

- l is a unique label for this constraint,
- c is a symbol denoting a constraint relation, and
- $v_1, \ldots, v_n$ are constraint variables, i.e., they can represent domain objects (including time points), variables in activities (which may have binding constraints attached).

Constraints represent the relations that must hold between the different objects related in the constraints for the plan to be executable. In the context of planning, the most commonly used constraints are of the following types:

- Ordering constraints: Let v_1, v_2 be variables in the plan representing time points. Then the constraint that v_1 has to be before v_2 can be represented as:
$$l : \texttt{before}(v_1, v_2)$$
- World-state constraints: Let p be a world-state proposition and v a variable representing a time point in the plan. Then the fact that p is a condition that has to hold at the time point represented by v, or the fact that p is an effect of an activity that holds at time point v can be represented respectively as:
$$l : \texttt{cond}(p, v)$$
$$l : \texttt{effect}(p, v)$$
- Variable binding constraints: Let v be a variable mentioned in some activity $a \in N$ and s be a constant symbol in the planning domain. Then the fact that v must take the value s can be represented as:
$$l : \texttt{value}(v, s)$$

These are just some of the constraint types that can be defined. The objects related to each other can be of different types. This is reflected by the domains of the constraint variables representing them. They can be world state propositions as in conditions and effects, or they can be variables used in activities representing time points or other domain objects in the plan as in ordering and variable binding constraints.

3.4. Annotations

Annotations are used to capture contextual information surrounding the current task. For more formal details on annotations and their use as a representation for rationale see [14].

4. Mental Attitudes in <I-N-C-A>

The idea behind augmenting <I-N-C-A> with mental attitudes is to focus on generating and executing the activities the agent performs. In essence, we want to define an agent that maintains an internal state [15] where the state can be represented by an <I-N-C-A> object, that is, a plan. As for all state-based agents, the state is updated based on the percepts the agent receives and the state determines the actions the agent performs. Given the above description of the state of an agent as an <I-N-C-A> object, in this section we shall describe how the components of this object relate to the components of the BDI view of agent state.

4.1. Beliefs/Constraints

The set of beliefs that an agent maintains corresponds reasonably directly to the constraints that hold in its current <I-N-C-A> object. The set of constraints includes facts about the agent's environment (including facts about other agents in that environment); it also includes task and domain knowledge in the form of ontological constraints on the environment and valid activity refinements. Also included among the beliefs is reasoning knowledge directly related to the planning task itself.

A set of *issue update rules* R_I are used to add new issues to the current plan following a revision to the set of beliefs/constraints. An issue update rule $r_I \in R_I$ is a triple (b, S, I) where:

- b is a newly asserted belief;
- S is an <I-N-C-A> description consisting of issues, activities, constraints and annotations, and;
- I is a set of issues that is to be added if the rule fires.

The idea here is that the belief b can be used to index relevant rules that may add issues when a new belief has been asserted. The <I-N-C-A> description S acts in effect as a precondition for the rule: all the components in S must be present in the agent's current plan for the rule to be applicable. Finally, if the rule fires the issue I will be added to the set of issues in the <I-N-C-A> model/plan of the agent.

4.2. Desires

There are a number of conflicting views of what constitutes a desire in the literature on BDI agents. For example, [15] describes them as the options the agent has, with the chosen option, the one to which it commits becoming an intention. This is not the view we will take here.

There is no explicit analogue to desires in this activity-centric model: desires are not explicitly represented in the <I-N-C-A> object state description of an agent. Instead, the desires of the agent can be considered *implicit* in elements of its reasoning about its current plan:

- the issue update rules described above are formulated according to the agent's desires. For example, if a new belief suggests a state of affairs counter to the agent's desires, an appropriate rule will cause an issue to be raised—this is a 'flaw' in the agent's plan.
- in a similar vein, the agent's desires are manifest in the 'utility function' which indicates which of a number of optional revisions to the plan should be selected to resolve a particular issue.

It is in these forms that desires appear in the algorithm that renders the agent specification executable (see section 5.4 below). One corollary of this notion of desires as latent in the agent's reasoning mechanisms is that any change in its desires can only be effected through the alteration of these mechanisms.

4.3. Intentions/Activities

Intentions in BDI are the actions an agent is committed to executing. In our activity-centric view, activity schemata instances (that is, the set of <I-N-C-A> activity nodes) in the agent's current state (<I-N-C-A> object) correspond directly to the BDI intentions of the agent—these are the activities that the agent is committed to performing. We shall now describe how these activity schemata and their instances are represented in our model.

We use the term *activity schema* to denote a type or class of activity, as distinct from instances of that activity. An activity schema $\mathcal{A}$ is a triple (s, C, E) where:

- s, the signature of the activity schema, is an expression of the form $n(v_1, \ldots, v_n)@[\tau_b, \tau_e]$, where:
  - n is the unique name of the activity schema;
  - $v_1, \ldots, v_n$ are variables representing parameters of the activity schema, and;
  - τ_b and τ_e are time points representing the beginning and end of an activity;
- C is the set of (pre-)conditions of the activity schema, where each $c_i \in C$ is either a state-variable expression of the form $S(v_{i,1}, \ldots, v_{i,k}) = o_i@\tau_i$ for state-variable $S(v_{i,1}, \ldots, v_{i,k})$, $v_{i,1}, \ldots, v_{i,k} \subseteq v_1, \ldots, v_n$, object constant o_i and relative time point τ_i, or it is a first-order literal $\pm P(v_{i,1}, \ldots, v_{i,k})@\tau_i$, and;
- E is the set of effects of the activity schema, where each $e_i \in E$ is either a state-variable assignment of the form $S(v_{i,1}, \ldots, v_{i,k}) \leftarrow o_i@\tau_i$ or it is a first-order literal $\pm P(v_{i,1}, \ldots, v_{i,k})@\tau_i$.

Thus, an activity schema corresponds to an operator schema in a STRIPS-like planning formalism. We designate an *activity* to be a partially instantiated activity schema; hence, an activity A is a pair (l, s) where:

- l is a unique label for this activity, and;
- s is an expression of the form $n(t_1, \ldots, t_n)@[\tau_b, \tau_e]$ such that there exists an activity schema $\mathcal{A}$ with signature $(n(v_1, \ldots, v_n)@[\tau'_b, \tau'_e]$ and a substitution σ such that for all i, $1 \leq i \leq n$: $\sigma(v_i) = t_i$.

Hence, each activity is an instance of some activity schema, and the activities in an agent's current <I-N-C-A> object correspond to its intentions. (Note that the unique label associated with an activity is necessary to distinguish between multiple instances of the same activity schema that may appear in the same <I-N-C-A> object.)

Finally, an *action* is a fully instantiated activity that can be executed by the agent. Hence, actions too correspond to the (subset of grounded) BDI intentions of the agent. Usually the performance of an action does not require further planning, with procedural knowledge available whose execution constitutes the performance of the action.

5. Interpreting <I-N-C-A> with BDI

In this section we shall describe the algorithms that take a specification of an activity-centric agent and produce the behaviour of that agent, i.e. they choose the actions the agent will perform. This description fits into the general "agent with state" model described in [15], in which an agent receives percepts, modifies its internal state accordingly, and chooses an action to execute based on its current state.

In our model of activity-centric agency, at any time τ the dynamic aspect of the internal state of an agent is described by the plan the agent intends to follow, where a plan is an <I-N-C-A> object as described above. Further knowledge of the agent includes its static desires that need not be included in the state. Let S_A be the current internal state of the agent A (I in [15]) and p the newly received percept. Then the overall algorithm the agent follows to compute the next internal state and executable action is as follows:

function actions(p) : 2^A
$\Delta Bel \leftarrow changedBeliefs(p, S_A)$
$Bel_A \leftarrow assert(\Delta Bel, Bel_A)$
$Iss_A \leftarrow propagateBeliefs(\Delta Bel, S_A)$
$Opts \leftarrow handleIssues(S_A)$
$S_A \leftarrow selectOption(Opts, Des_A)$
$Acts \leftarrow getExecutables(S_A)$
return $Acts$

The first difference between this function and the standard $action$-function is that it returns a set of actions (2^A denotes the powerset of all possible actions). This is simply because the I-X planner, like most planners, generates plans that may contain parallel actions. The executing agent may also be able to perform actions in parallel. If the agent cannot execute the returned actions concurrently, it needs to perform them sequentially. For independent actions this should not change the eventual world state.

The first step of the algorithm computes the changes to the current set of beliefs that result from the new percept p and stores them in ΔBel, a local variable. Next the changed beliefs are asserted into the current state, at this stage only modifying the beliefs. The next step then uses the issue update rules R_I and updates the current state of the agent using the function *propagateBeliefs*. This may create new issues in the current plan of the agent which need to be addressed before further action can be taken. The function *handleIssues* does this using the HTN planner, which creates at least one new plan. If there is more than one plan, the function *selectOption* chooses the one that will be adopted based on the desires of the agent. Finally, the set of currently executable actions is extracted from the current plan and returned.

5.1. Updating the Beliefs

The first two steps in the algorithm above are concerned with the updating of the set of beliefs currently held by the agent. There is little that can be said at this level about the implementation of the functions *changedBeliefs* and *assert*. The first function computes the set of atoms in the dynamic belief set that will have their value changed. This may be different from the percepts (which include communication with other agents) as trust needs to be taken into account and inconsistencies with other knowledge must be prevented. Asserting the changed beliefs simply modifies the current beliefs Bel_A according to the computed changes.

The resulting state of the agent at this point is that its beliefs have been modified according to the given percept and a trace of the changes is available in ΔBel.

5.2. Applying Issue Update Rules

Now the function *propagateBeliefs* uses the new beliefs to check whether any of the issue update rules will add new issues to the current plan. This can be done as follows:

function propagateBeliefs($\Delta Bel, S_A$) : Iss_A
$issues \leftarrow \emptyset$
for every $b_{new} \in \Delta Bel$ **do**
if $\exists r = (b_r, S_r, I_r) \in R_I$ **and** $\exists \sigma$: substitution
and $\sigma(b_r) = b_{new}$
and $\sigma(S_r) \subseteq S_A$ **then**
$issues \leftarrow issues \cup \sigma(I_r)$
return $issues$

This function accumulates the new issues that are generated by all the newly asserted beliefs in the variable $issues$ which is returned at the end. Each new belief is considered in turn. If there is an issue update rule that has a matching belief and the <I-N-C-A> components of that rule are part of the current state then a new issue is added to the result. The new issue is instantiated using the same substitution that was the result of matching the belief and the <I-N-C-A> preconditions.

The resulting state of the agent at this point is that its beliefs are updated and new issues with the current plan arising from these modified beliefs have been identified.

5.3. Handling Issues

As mentioned in section 3.1, issues can be seen as meta-level activities, things that need to be done to the current plan before it becomes a solution to the current planning problem. Hence the function *handleIssues* modifies the current plan such that it will not contain issues. Since there may be more than one way of achieving this, this function returns a set of possible plans. The function that does this works as follows:

function handleIssues(S_A) : 2^{S_A}
  $options \leftarrow getPlans(S_A)$
  if $options = \emptyset$ **then**
    $S' \leftarrow resetProblem(S_A)$
    $options \leftarrow getPlans(S')$
  while $options = \emptyset$ **do**
    $S' \leftarrow dropIntention(S')$
    $options \leftarrow getPlans(S')$
  return $options$

This function first attempts to add <I-N-C-A> components to the current plan that will resolve all issues. This is achieved by calling the function *getPlans* which is the main function of the HTN refinement planner in I-X. This will use specific issue handlers that remove the issue they are addressing from the plan and add other <I-N-C-A> components to to it, including new issues. Often there is more than one specific issue handler applicable, or a handler may be applicable in multiple ways. As a result, the planner generates a set of plans that constitute the options at this point. Ideally, the planner would generate a small, but interestingly different set of options.

In the worst case the set of generated options is empty, meaning the planner was unable to find a way to address all the issues in the current plan and come up with a workable course of action. In this case the function attempts to replan, rather than modify the existing plan. To do this, it first needs to create a new planning problem that consists of the current state and only the most abstract intentions in the current task network with their constraints. New issues that express the need for refinement must be added before the planner can again be invoked to attempt to find a plan.

Again, this may not succeed. That is, neither plan modification nor replanning can be used to generate a plan that implements all the current intentions. In this case the only way to continue is to start dropping intentions until a workable plan can be found, and this is what the while-loop in the above procedure does. It takes the same planning problem that was used for replanning and selects an intention to remove. This can be done by priority and analyzing the reasons for failure. Eventually, the planner has a problem that contains activities that can be refined into one or more solution plans. At this point the generated options are returned.

The resulting state of the agent is that its beliefs are updated and resulting issues have been addressed giving the agent a number of plans as options.

5.4. Selecting an Option

The function *selectOption* now has to choose the preferred option for the agent to adopt as its next internal state.

function selectOption($Opts, Des_A$) : S_A
  $best \leftarrow -\infty$
  for every $p \in Opts$ **do**
    $ws \leftarrow project(Bel_A, p)$
    $util \leftarrow evaluate(ws, p, Des_A)$
    if $util > best$ **then**
      $plan \leftarrow p$
      $best \leftarrow util$
  return $plan$

Essentially, what this function does is project the outcome of every option that is currently being considered into a new world state that would be the result of executing that plan. This projected world state is stored in the variable ws. Then the function computes the utility of that state, also taking into account the plan that produced it, and the desires as described in section 4.2, i.e. in form of a utility function. The result is a projected utility of the option with respect to the agent's desires. The function then simply remembers the best option (in the variable $plan$) and its utility (in the variable $best$) and returns this best option.

The resulting state of the agent at this point is that both its beliefs and its plan are updated.

5.5. Executable Actions

The function that computes the next executable actions is quite simple for activity-centric agents. Since the internal state already represents the plan the agent intends to follow, the next executable actions are simply those that have no predecessors in the current plan and for which the current time lies within the time interval in which the actions are intended to be performed. The result is a set of executable actions. Since there are no ordering constraints between these actions they must also be independent, meaning they can be executed in any order. The reason why they are not serialized here is simply that we do not want to assume that the agent can only perform one of them at a time.

Now the agent can execute the chosen actions and this concludes the perceive-deliberate-act cycle for activity-centric agents.

6. Conclusions and Future Work

In this paper we have presented a new agent framework that reconciles I-X, with its powerful HTN planner and its underlying <I-N-C-A> representation, with the mental attitudes of the BDI model. Agents conforming to this new framework adopt an activity-centric view of a situation, focusing on what needs to be done, a natural consequence of the origins of the framework in planning and, more specifically, of the representation of the agent's internal state as a plan. With the introduction of BDI attitudes into <I-N-C-A>, intentions are now represented as activities at various levels of abstraction, while desires are not merely another type of activity (as e.g. in [3]) but instead are used to decide which of a number of possible courses of action the agent will adopt. Thus, they are part of the decision-making knowledge of the agent. The semantics of this representation is defined through procedures that describe how the specification of the agent, together with the percepts received, result in executable actions.

The augmentation of the <I-N-C-A> model with beliefs, desires and intentions results in a new activity-centric framework that combines the advantages of both HTN planning and BDI agency. Virtually all practical planners are based on the HTN approach to which the <I-N-C-A> model conforms. Given a set of activity refinements, which correspond naturally to procedural domain expertise, and a high-level task specification, HTN planners are capable of generating plausible, annotated (e.g., with rationale) task networks consisting of executable actions that achieve the specified tasks. This ability to reason about actions is the strength of HTN planning. From a BDI perspective, this provides a practical and proven mechanism for the generation of options that is a vital step in the BDI agent's inference loop. The BDI model has become the *de facto* standard for representing the mental state of an agent. It provides a clear delineation of the different types of knowledge an agent requires, and the role each type plays in the agent's inference mechanisms. While HTN planners are capable of generating activity options, the mental attitudes of the BDI approach, as manifest in its knowledge structures, provide the means of selecting among these options, and committing to a particular course of action. In this way, these attitudes provide a decision-making level that is not considered in the traditional formulation of the planning problem.

In summary, this fusion of HTN planning and the mental attitudes of BDI results in a new agent framework in which the two paradigms are seen to be complementary: HTN planning provides a competence required in the BDI inference loop, and BDI provides a description of the knowledge necessary to move from planning to acting. In the future, we intend to extend our framework to include agent social abilities (interaction, cooperation, etc.), and to investigate the concept of commitments [5] and their representation in <I-N-C-A>. The aim in extending the work in this direction is to support distributed planning among a collective of decentralized and fully autonomous yet collaborating entities.

References

[1] J. Allen, J. Hendler, and A. Tate, eds. *Readings in Planning*. Morgan Kaufman, 1990.

[2] K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[3] M. Georgeff and A. Lansky. Reactive reasoning and planning. In *Proc. National Conf. of the American Association of Artificial Intelligence (AAAI)*, pp. 677–682. AAAI Press, 1987.

[4] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning*. Morgan Kaufmann, 2004.

[5] N. R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review*, 8(3):223–250, 1993.

[6] A. Rao and M. Georgeff. Modeling rational agents within a BDI-architecture. In *Proc. 2nd Int. Conf. on Knowledge Representation and Reasoning (KR)*, pp. 473–484. Morgan Kaufmann, 1991.

[7] E. D. Sacerdoti. The nonlinear nature of plans. In *Proc. 4th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pp. 206–214. Morgan Kaufmann, 1975. Reprinted in [1, pp. 162–170].

[8] S. Sardina, L. de Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. 5th Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1001–1008. ACM Press, 2006.

[9] S. Smith. OPIS: A methodology and architecture for reactive scheduling. In M. Zweben and M. Fox, eds, *Intelligent Scheduling*. Morgan Kaufmann, 1994.

[10] A. Tate. Generating project networks. In *Proc. 5th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pp. 888–893. Morgan Kaufmann, 1977. Reprinted in [1, pp. 291–296].

[11] A. Tate. Intelligible AI planning. In M. Bramer, A. Preece, and F. Coenen, eds, *Research and Development in Intelligent Systems XVII (Proc. 20th ES)*, pp. 3–16. Springer, 2000.

[12] A. Tate. <I-N-C-A>: A shared model for mixed-initiative synthesis tasks. In G. Tecuci, editor, *Proc. IJCAI Workshop on Mixed-Initiative Intelligent Systems*, pp. 125–130, 2003.

[13] A. Tate, J. Dalton, and J. Levine. O-Plan: A web-based AI planning agent. In *Proc. National Conf. of the American Association of Artificial Intelligence (AAAI)*, pp. 1131–1132. AAAI Press, 2000.

[14] G. Wickler, S. Potter, and A. Tate. Recording rationale in <I-N-C-A> for plan analysis. In L. McCluskey, K. Myers, and B. Srivastava, eds, *Proc. ICAPS Workshop on Plan Analysis and Management*, pp. 5–11, 2006.

[15] M. Wooldridge. Intelligent agents. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pp. 27–77. The MIT Press, 1999.

[16] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.