Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach*

Sebastian Sardina
RMIT University
Melbourne, Australia
ssardina@cs.rmit.edu.au

Lavindra de Silva
RMIT University
Melbourne, Australia
ldesilva@cs.rmit.edu.au

Lin Padgham
RMIT University
Melbourne, Australia
linpa@cs.rmit.edu.au

ABSTRACT

This paper provides a general mechanism and a solid theoretical basis for performing planning within Belief-Desire-Intention (BDI) agents. BDI agent systems have emerged as one of the most widely used approaches to implementing intelligent behaviour in complex dynamic domains, in addition to which they have a strong theoretical background. However, these systems either do not include any built-in capacity for "lookahead" type of planning or they do it only at the implementation level without any precise defined semantics. In some situations, the ability to plan ahead is clearly desirable or even mandatory for ensuring success. Also, a precise definition of how planning can be integrated into a BDI system is highly desirable. By building on the underlying similarities between BDI systems and Hierarchical Task Network (HTN) planners, we present a formal semantics for a BDI agent programming language which cleanly incorporates HTN-style planning as a built-in feature. We argue that the resulting integrated agent programming language combines the advantages of both BDI agent systems and hierarchical offline planners.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents,Languages and structures*

Keywords

BDI agent-oriented programming, HTN planning

1. INTRODUCTION

The BDI (Belief-Desire-Intention) model is a popular and well-studied architecture of agency for intelligent agents situated in complex and dynamic environments. The model has its roots in philosophy with Bratman's [2] theory of practical reasoning and Dennett's theory of intentional systems [9]. There are a number of agent programming languages in the BDI tradition, such as PRS [13], AGENTSPEAK [19], 3APL [12], JACK [3], and CAN [24].

BDI agent-oriented systems are extremely flexible and responsive to the environment, and as a result, well suited for complex applications with real-time reasoning and control requirements. However, a limitation of these systems is that they normally do no *lookahead* or *planning* in the traditional sense; execution is based on a user-provided "plan library" to achieve goals. BDI frameworks rely entirely on context sensitive subgoal expansion, acting as they go. In some circumstances, however, lookahead deliberation (i.e., hypothetical reasoning) about the effects of one choice of expansion over another is clearly desirable, or even mandatory in order to guarantee goal achievability and to avoid undesired situations. In general, this is the case when (a) important resources may be used in taking actions that do not lead to a successful outcome; (b) actions are not always reversible and may lead to states from which there is no successful outcome; (c) execution of actions take substantially longer than "thinking" (or planning); and (d) actions have side effects which are undesirable if they turn out not to be useful.

In this paper, we develop a traditional BDI-style agent programming language that includes an *on-demand planning mechanism* in the style of Hierarchical Task Networks (HTN), whose semantics and implementations are well understood in the planning community [11]. The language we propose, named CANPLAN, provides a flexible approach regarding when to perform full lookahead and is provably more expressive than either BDI or HTN systems alone. CANPLAN is based on CAN [24] and AGENTSPEAK [19]. One could argue, of course, that it is always possible, in critical situations, to explicitly program lookahead within existing BDI systems. However, such code would generally be domain dependent, can be fairly complex, and would lie outside the infrastructure support provided by the BDI agent platform. Alternatively, there are many frameworks that attempt to interleave BDI-type execution with offline planning (e.g., [1, 23, 10, 17, 14]). Still, these are mostly implemented systems with no precise semantics and with little programmer control over when to plan. Our approach, instead, is to provide a formal specification of planning as a *built-in feature* of the BDI infrastructure that the programmer can use as appropriate.

The contributions of this paper are threefold. Firstly, a precise account of planning within a typical BDI agent programming language is provided. Secondly, the intrinsic relationship between lookahead planning in the context of BDI agents and the HTN approach to planning is formally explored. Lastly, the semantics of CAN given in [24] is substantially improved and simplified.

The rest of the paper is organised as follows. In section 2, we provide a brief overview of BDI agent programming languages and HTN planners; we also provide an informal discussion on their similarities. In section 3, we describe the basic BDI agent language we will use, namely the CAN notation described in [24], but with some modifications to include actions with preconditions and ef-

fects, multiple variable bindings, and a simpler, though equivalent, account of declarative goals. We chose CAN from the numerous available options because it has the desirable features of (a) combining a declarative and procedural view of *goals*, and (b) capturing the semantics of BDI failure recovery and goal persistence. In section 4, we develop CANPLAN, our new integrated account of planning and BDI execution. Besides showing some intuitively expected properties for the combined framework, we prove that, under suitable assumptions, CANPLAN's planning module reduces to HTN planning. In section 5, a brief discussion on a prototype implementation is given. Section 6 discusses related work. Finally, in section 7, we draw conclusions and outline future lines of research.

2. BDI AND HTN SYSTEMS

There are a number of BDI agent languages and HTN systems. In this section, we provide a brief abstract overview of these and comment on their similarities.

2.1 BDI Agent Programming Languages

Generally speaking, BDI agent-oriented programming languages are built around an explicit representation of beliefs, desires, and intentions. A BDI architecture addresses how these components are represented, updated, and processed to determine the agent's actions. There are a substantial number of implemented BDI systems, as well as a number of formally specified languages.

An agent consists, basically, of a belief base $\mathcal{B}$, a set of recorded pending events (goals), a plan library Π, and an intention base Γ. The *belief base* encodes the agent's knowledge about the world. The *plan library* contains *plan rules* of the form $e : \psi \leftarrow P$ encoding a *plan-body program* P for handling an event-goal e when *context* condition ψ is believed to hold. The *intention base* contains the current, partially instantiated, plans the agent has already committed to in order to handle or achieve some event-goal.

A BDI system responds to *events*, the inputs to the system, by committing to handle one pending event-goal, selecting a plan rule from the library, and placing its plan-body program into the intention base. The execution of this program may, in turn, post new sub-goal events to be achieved. If at any point a program fails, then an alternative plan rule is found and its plan-body is placed into the intention base for execution. This process repeats until a plan succeeds completely or until there are no more applicable plans, in which case failure is propagated to the event-goal.

In section 3, we shall discuss in detail one formal BDI language of this sort, namely, the CAN language [24].

2.2 HTN Planning

Hierarchical Task Network (HTN) planning is an approach to planning based on the decomposition of (high-level) *tasks* in order to accomplish an (initial) *task network*. Two examples of HTN systems include SHOP [15] and its successor SHOP2 [16]. Below, we mostly follow the definitions of HTN-planning from [11].

Tasks can be of two types. A *primitive task* is an action $act(\vec{x})$ that can be directly executed by the agent (e.g., $drive(x_1, x_2)$). A (high-level) *compound task* $e(\vec{x})$ is one that cannot be executed directly (e.g., $build\_trip(origin, dest)$). A *task network* $d = [T, \phi]$ is a collection of tasks T that need to be accomplished and a boolean formula of constraints ϕ. Constraints impose restrictions on the ordering of the tasks ($e \prec e'$), on the binding of variables ($x = x'$) and ($x = c$) (c is a constant), and on what literals must be true before or after each task (l, e), (e, l), and (e, l, e'). A *method* (e, ψ, d) encodes a way of decomposing a high-level compound task e into lower-level tasks using task network d when ψ holds. Methods provide the procedural knowledge of the domain.

An HTN *planning domain* $\mathcal{D} = (\Pi, \Lambda)$ consists of a library Π of methods and a library Λ of primitive tasks. Each primitive task in Λ is a STRIPS style action with corresponding preconditions and effects in the form of *add* and *delete* lists. An HTN *planning problem* **P** is a triple $\langle d, \mathcal{B}, \mathcal{D} \rangle$ where d is the task network to accomplish, $\mathcal{B}$ is the initial state (i.e., a set of all ground atoms that are true in $\mathcal{B}$), and $\mathcal{D}$ is a planning domain. A *plan* σ is a sequence $act_1 \cdot \ldots \cdot act_n$ of ground actions (that is, ground primitive tasks).

Given a planning problem instance **P**, the planning process involves selecting and applying an applicable reduction method from $\mathcal{D}$ to some compound task in d. This results in a new, and typically more "primitive," task network d'. This reduction process is repeated until only primitive tasks (i.e., actions) remain. If no applicable reduction can be found for a compound task at any stage, the planner "backtracks" and tries an alternative reduction for a compound task previously reduced. If all compound tasks can eventually be reduced, a plan solution σ is obtained

In [11], a clear operational semantics for HTN planning was given. The set of plans $sol(d, \mathcal{B}, \mathcal{D})$ that solves a planning instance $\mathbf{P} = \langle d, \mathcal{B}, \mathcal{D} \rangle$ is defined as $sol(d, \mathcal{B}, \mathcal{D}) = \bigcup_{n < \omega} sol_n(d, \mathcal{B}, \mathcal{D})$, where $sol_n(d, \mathcal{B}, \mathcal{D})$ is, in turn, defined as follows:

$$
\begin{aligned}
sol_1(d, \mathcal{B}, \mathcal{D}) &= comp(d, \mathcal{B}, \mathcal{D}), \\
sol_{n+1}(d, \mathcal{B}, \mathcal{D}) &= sol_n(d, \mathcal{B}, \mathcal{D}) \cup \bigcup_{d' \in red(d, \mathcal{B}, \mathcal{D})} sol_n(d', \mathcal{B}, \mathcal{D}).
\end{aligned}
$$

Intuitively, $comp(d, \mathcal{B}, \mathcal{D})$ is the set of all plan *completion* of a network d containing only primitive tasks (i.e., plans for which the constraint formula ϕ in d is satisfied), and $red(d, \mathcal{B}, \mathcal{D})$ is the set of all *reductions* of d in $\mathcal{B}$ by methods in $\mathcal{D}$. We refer to [11] for more details on HTN and its formal semantics.

2.3 Similarities Between HTN and BDI

As stated in [7], BDI agent programming languages and HTN planners share many similarities despite their different purposes. The similarities come from the knowledge used by both systems as well as from how this knowledge is manipulated to create solutions.

First of all, HTN systems and BDI languages assume an explicit representation of the agent's knowledge (i.e., the belief base or state) and a set of primitive tasks or actions that the agent can directly execute in the world. Secondly, procedural knowledge about the domain is available in both HTN and BDI systems in the form of reduction methods and plan rules, respectively. Thirdly, and most importantly, both systems create solutions by reducing higher-level entities into lower-level ones by appealing to a given set of reduction recipes. Whereas a BDI system "reduces" an event into an plan-body program using a plan rule from the plan library, an HTN planner reduces a compound task into a task network using a reduction method from the method library.

The following table gives an indication of the mapping between HTN and BDI entities.

BDI SYSTEMS	HTN SYSTEMS
belief base	state
plan library	method library
event	compound task
action	primitive task
plan-body/program	network task
plan rule	method
plan rule context	method precondition
test ?l in plan-body	state constraints
sequence in plan-body	ordering constraint $\prec$
parallelism in plan-body	no ordering constraint

The above table is not complete—whereas some entities have a straightforward mapping some others require a more elaborate translation (we refer to [7, 22] for a more detailed mapping).

BDI agent systems and HTN planners, despite their close relationship, differ fundamentally in their objectives. The former are focused on the *execution* of agent programs where "backtracking" can only happen in the real world. The latter, in contrast, are concerned with *hypothetical reasoning* about actions and their potential interactions within a whole plan for achieving a goal or task.

3. THE BASIC BDI LANGUAGE

The CAN (**C**onceptual **A**gent **N**otation) notation [24] is a high-level plan language in the style of typical agent languages, both in the BDI tradition and elsewhere (e.g., AGENTSPEAK [19], 3APL [12, 21], and even CONGOLOG [5, 6]). Its syntax and semantics attempt to extract the essence of a class of implementable agent platforms and could be considered as a superset of AGENTSPEAK. Unlike AGENTSPEAK, though, the semantics for CAN includes both *failure handling* and *declarative goals*—two appealing features for our planning agents.

An agent is created by the specification of a set of base beliefs $\mathcal{B}$ and a set of plans Π. The *belief base* of an agent is a set of formulas from some (knowledge representation) logical language. The programmer may choose any logical language; all that is required is for operations to exist that check whether a condition ϕ—a logical formula over the agent's beliefs—follows from a belief set (i.e., $\mathcal{B} \models \phi$), and to add and delete a belief b to and from a belief base (i.e., $\mathcal{B} \cup \{b\}$ and $\mathcal{B} \setminus \{b\}$, respectively). In practice, however, the belief base contains ground belief *atoms* in a first-order language.

As explained in section 2, an agent *plan library* Π consists of a collection of plan rules of the form $e : \psi \leftarrow P$, where e is an event and ψ is the context condition which must be true in order for the plan-body P to be applicable.[1] The *plan-body* or *program* P is built from primitive actions act that the agent can execute directly, operations to add $+b$ and delete $-b$ beliefs, tests for conditions $?\phi$, and events or (internal) achievement goals $!e$. Complex plans can be specified using sequencing $P_1 ; P_2$, parallelism $P_1 \| P_2$, and declarative goals $\mathsf{Goal}(\phi_s, P, \phi_f)$ (explained later). Hence, the *user language* is described by the following grammar:

$$P ::= act \mid +b \mid -b \mid ?\phi \mid !e \mid P_1; P_2 \mid P_1 \| P_2 \mid \mathsf{Goal}(\phi_s, P_1, \phi_f).$$

There are also a number of auxiliary plan forms which are used internally when assigning semantics to constructs: basic (terminating) program *nil*; and compound plans $P_1 \triangleright P_2$, which executes P_1 and then executes P_2 only if P_1 failed; and $(\![\psi_1 : P_1, \ldots, \psi_n : P_n]\!)$, which is used to encode a set of (relevant) guarded plans. The *full language* is therefore described by the following grammar:

$$P ::= nil \mid act \mid ?\phi \mid +b \mid -b \mid !e \mid P_1; P_2 \mid P_1 \triangleright P_2 \mid P_1 \| P_2 \mid$$
$$\mathsf{Goal}(\phi_s, P_1, \phi_f) \mid (\![\psi_1 : P_1, \ldots, \psi_n : P_n]\!).$$

In contrast with [19, 24], we take actions as the usual basic means of the agent to change its environment and, hence, actions may have preconditions and effects. One possibility would be to follow [12] and assume that a *partial* function $\mathcal{T}$ specifying the update semantics of basic actions is given: if $\mathcal{T}(act, \mathcal{B})$ is defined, it yields the new updated belief base $\mathcal{B}'$; otherwise, we say that the action's precondition is not met in $\mathcal{B}$. However, for simplicity, we shall restrict ourselves to agents that are equipped with a (simple) STRIPS-like *action description library* Λ containing rules of the form $act : \psi_{act} \leftarrow \Phi^-_{act}; \Phi^+_{act}$, one for each action type in the domain. Formula ψ_{act} corresponds to the action's precondition, and Φ^+_{act} and Φ^-_{act} stand for the add and delete lists of atoms, respectively.[2] For example, action $move(x, y, z)$, which moves object x

---

[1] An omitted ψ is equivalent to *true*. Notice that e, ψ, and P may contain free variables; a plan rule is of the form $e(\vec{x}) : \psi(\vec{x}, \vec{y}) \leftarrow P(\vec{x}, \vec{y}, \vec{z})$, where $\vec{x}, \vec{y}$ and $\vec{z}$ are vectors of (distinct) variables.

[2] Free variables in ψ_{act}, Φ^-_{act} and Φ^+_{act} are free in act too.

from y to z, could be represented in Λ as follows:

$$move(x, y, z) : Free(z) \wedge At(x, y) \longleftarrow$$
$$\{Free(z), At(x, y)\}; \{Free(y), At(x, z)\}.$$

Next, we show the operational semantics for the above language along the lines of [24]. A *transition relation* $\longrightarrow$ on so-called *configurations* is defined by a set of derivation rules. A transition $C \longrightarrow C'$ specifies that executing configuration C a *single step* yields configuration C'. We write $C \longrightarrow$ to state that there exists C' such that $C \longrightarrow C'$, and $\stackrel{*}{\longrightarrow}$ to denote the usual reflexive transitive closure of $\longrightarrow$. A derivation rule consists of a, possibly empty, set of premises, which are transitions together with some auxiliary conditions, and a single transition conclusion derivable from these premises. (see [18] for more on operational semantics).

Two types of transitions will be used to define the semantics of our agents. The first type defines what it means to execute a single intention and is defined in terms of *basic* configurations. The second type of transition is defined in terms of the first type and defines what it means to execute an agent. A <u>*basic configuration*</u> is a tuple $\langle \mathcal{B}, \mathcal{A}, P \rangle$ consisting of the current belief base $\mathcal{B}$ of the agent, the sequence $\mathcal{A}$ of primitive actions executed so far, and the plan-body program P being executed (i.e., the current intention).[3]

Here are some of the core derivation rules for the language:

$$\frac{\Delta = \{\psi_i \theta : P_i \theta \mid e' : \psi_i \leftarrow P_i \in \Pi \wedge \theta = \mathsf{mgu}(e, e')\}}{\langle \mathcal{B}, \mathcal{A}, !e \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, (\![\Delta]\!) \rangle} \; Event$$

$$\frac{\psi_i : P_i \in \Delta \quad \mathcal{B} \models \psi_i \theta}{\langle \mathcal{B}, \mathcal{A}, (\![\Delta]\!) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P_i \theta \triangleright (\![\Delta \setminus P_i]\!) \rangle} \; Sel$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1 \rangle \stackrel{bdi}{\not\longrightarrow}}{\langle \mathcal{B}, \mathcal{A}, (P_1 \triangleright P_2) \rangle \stackrel{bdi}{\longrightarrow} \langle \mathcal{B}, \mathcal{A}, P_2 \rangle} \; \triangleright_f \quad \frac{\mathcal{B} \models \phi \theta}{\langle \mathcal{B}, \mathcal{A}, ?\phi \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil \rangle} \; ?$$

$$\frac{a : \psi \leftarrow \Phi^-; \Phi^+ \in \Lambda \quad a\theta = act \quad \mathcal{B} \models \psi \theta}{\langle \mathcal{B}, \mathcal{A}, act \rangle \longrightarrow \langle (\mathcal{B} \setminus \Phi^- \theta) \cup \Phi^+ \theta, \mathcal{A} \cdot act, nil \rangle} \; act$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P' \rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1; P_2) \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', (P'; P_2) \rangle} \; Seq$$

Rule $Event$ handles achievement goal events by collecting all *relevant* plans for the event in question. Rule Sel selects one *applicable* plan from a set of (remaining) relevant plans: program $P \triangleright (\![\Delta]\!)$ states that program P should be tried first, falling back to the remaining alternatives in Δ if necessary. Notice that plan rules' context conditions are handled in a lazy manner. Rule $?$ deals with test goals by checking that the condition follows from the current belief base, whereas rule act handles the case of primitive actions by using the domain action description library Λ. Rule Seq handles sequencing of programs in the usual way. Rule $\triangleright_f$ is used along with rule Sel for *failure handling*: if the current plan $P_i \theta$ for a goal fails (e.g., at some point the precondition of an action or a test goal is not met), rule $\triangleright_f$ applies first, and eventually, rule Sel may select another *applicable* alternative for the event-goal, if any.

A central distinguishing feature of CAN is its $\mathsf{Goal}(\phi_s, P, \phi_f)$ goal construct, which provides a mechanism for representing both declarative and procedural aspects of goals. Intuitively, a *goal-program* $\mathsf{Goal}(\phi_s, P, \phi_f)$ states that we should achieve the (declarative) goal ϕ_s by using (procedural) program P; failing if ϕ_f becomes true. The execution of a goal-program is consistent with

---

[3] Strictly speaking, the plan and action libraries Π and Λ should also be part of basic configurations. For legibility purposes, we omit them as they are assumed to be static entities. Configurations must also include a variable substitution θ for keeping track of all bindings done so far during the execution of a plan-body. Again, for legibility, we keep substitutions implicit in places where they need to be carried across multiple rules. See [12] on how substitutions are propagated across derivation rules for 3APL.

some desired properties of declarative goals (namely, persistent, possible, and unachieved). For instance, if P is fully executed but ϕ_s is still not true, P will be re-tried; and if ϕ_s becomes true during P's execution, the whole goal will succeed immediately.

In order to capture the desired behaviour of goal-programs, a sophisticated operational semantics was given in [24], based on explicit exceptions, a set of conditions being "watched," and derivation rules with priorities. Here, we provide an alternative much *simpler* semantics that is equivalent to the original one. The following is the new set of rules for goal-programs:

$$\frac{P \neq P_1 \rhd P_2 \quad \mathcal{B} \not\models \phi_s \vee \phi_f}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P, \phi_f) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P \rhd P, \phi_f) \rangle} \; \mathsf{G}_I$$

$$\frac{\mathcal{B} \models \phi_s}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P, \phi_f) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil \rangle} \; \mathsf{G}_s$$

$$\frac{\mathcal{B} \models \phi_f}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P, \phi_f) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, ?false \rangle} \; \mathsf{G}_f$$

$$\frac{P = P_1 \rhd P_2 \quad \mathcal{B} \not\models \phi_s \vee \phi_f \quad \langle \mathcal{B}, \mathcal{A}, P_1 \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P' \rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P, \phi_f) \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', \mathsf{Goal}(\phi_s, P' \rhd P_2, \phi_f) \rangle} \; \mathsf{G}_S$$

$$\frac{P = P_1 \rhd P_2 \quad \mathcal{B} \not\models \phi_s \vee \phi_f \quad \langle \mathcal{B}, \mathcal{A}, P_1 \rangle \not\longrightarrow}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P, \phi_f) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P_2 \rhd P_2, \phi_f) \rangle} \; \mathsf{G}_R$$

When a goal-program is first encountered during execution, rule G_I applies: G_I "initialises" the execution of a goal-program by setting the program in the goal to $P \rhd P$, where the first P is to be executed and the second P is just used to store and carry the *original* program P in case rule G_R may eventually apply later on. The second and third rules handle the cases where either the success condition ϕ_s or the failure condition ϕ_f become true. The fourth rule G_S is the one responsible for performing a single step on an already initialised goal-program. Notice that the second part in the pair $P_1 \rhd P_2$ remains constant. Finally, rule G_R restarts the original program (stored as the second program in pair $P_1 \rhd P_2$) whenever the current program has finished or got stuck, but the desired, and still possible, goal has not been achieved yet.

The above semantics for Goal is substantially simpler than the original one in [24] in that we do not appeal to explicit exceptions, "watched" conditions, or special prioritised derivation rules. Although it is not hard to prove that this alternative semantics is equivalent to the original one, due to lack of space, we do not do that here. Finally, we point out that, in the original semantics of CAN, an agent also included a goal base $\mathcal{G}$ to account for the declarative goals the agent has already committed to via goal-programs. Although not done in CAN, the goal base could potentially be used to perform (meta)reasoning about goals at the agent level execution, such as goal conflict detection and resolution ([20]). Since we are also not concerned in this paper with this type of reasoning, we completely omit the goal base from our agents.

Agent Level Execution

On top of the above basic rules, we define the evolution of an agent. An *agent configuration*, or just an agent, is a tuple of the form $\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle$ where $\mathcal{N}$ is the agent name, Λ is an action description library, Π is a plan library, $\mathcal{B}$ is a belief base, $\mathcal{A}$ is the sequence of actions already performed by the agent, and Γ is the set of current intentions (that is, plan-body programs). Transitions between agent configurations are dictated by the following three rules:

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P' \rangle}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}', \mathcal{A}', (\Gamma \setminus \{P\}) \cup \{P'\} \rangle} \; A_{step}$$

$$\frac{e \text{ is a new external event}}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{!e\} \rangle} \; A_{event}$$

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P \rangle \not\longrightarrow}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{P\} \rangle} \; A_{clean}$$

The first rule performs a single step in one intention; the second rule creates a new intention from an external event; and the last rule removes a completed intention from the intention base, that is, an intention *nil* or one that is blocked and cannot make a transition.

Next, we define the meaning of an agent execution and two related notions that will be used later in the paper.

DEFINITION 1 (BDI EXECUTION). *A BDI <u>execution</u> E of an agent $C_0 = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}_0, \mathcal{A}_0, \Gamma_0 \rangle$ is a, possibly infinite, sequence of agent configurations $C_0 \cdot C_1 \cdot \ldots C_n \cdot \ldots$ such that $C_i \Longrightarrow C_{i+1}$, for every $i \geq 0$. A <u>terminating execution</u> is a finite execution $C_0 \cdot \ldots \cdot C_n$ where $C_n = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}_n, \mathcal{A}_n, \{\} \rangle$. An <u>environment-free</u> execution is one in which rule $\mathcal{A}_{event}$ has not been used.*

Sometimes we will be only interested in those steps of an execution where changes occur in either the executed actions or the belief of the agent—agent steps where the belief base and the executed actions remain unchanged can be disregarded. So, if $E = C_0 \cdot \ldots \cdot C_n$ is a (finite) execution, then the <u>derived execution</u> $\overline{E}$ is the sequence of configurations obtained from E by deleting all configurations C_j in the sequence such that $\mathcal{B}_j = \mathcal{B}_{j+1}$ and $\mathcal{A}_j = \mathcal{A}_{j+1}$.

In addition, we give the following notation to track an intention during an execution. If $C_0 \cdot \ldots \cdot C_n$ is a normal or derived execution and P is an intention in C_0 (i.e., $P \in \Gamma_0$), then the sequence $P_0 = P, P_1, \ldots, P_n$ denotes P's evolution within the execution and either (i) $P_i \in \Gamma_i$; or (ii) $P_i = \epsilon$, if the intention has already been removed from the intention base at some C_j, where $j \leq i$.

DEFINITION 2. *Two, possibly derived, agent executions $C_0 \cdot \ldots \cdot C_n$ and $C_0' \cdot \ldots \cdot C_n'$ are <u>equivalent modulo intentions</u> iff $C_i' = \langle \mathcal{N}_i, \Lambda_i, \Pi_i, \mathcal{B}_i, \mathcal{A}_i, \Gamma_i' \rangle$, for every $0 \leq i \leq n$. Also, the two executions are <u>equivalent modulo intentions</u> $P_0 \in \Gamma_0$ and $P_0' \in \Gamma_0'$ if they are equivalent modulo intentions and for every $0 \leq i \leq n$, $(\Gamma_i' \setminus \{P_i'\}) = (\Gamma_i \setminus \{P_i\})$ (where P_i (P_i') is P_0's (P_0''s) evolution in configuration C_i (C_i')).*

Lastly, we define what we mean by the execution of an intention and by a program (weakly) simulating another program.

DEFINITION 3 (INTENTION EXECUTION). *Let E be a BDI execution $C_0 \cdot C_1 \cdot \ldots \cdot C_n$ for an agent $C_0 = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}_0, \Gamma_0 \rangle$, where $\Gamma_0 = \Gamma_0' \cup \{P_0\}$. Intention P_0 in C_0 has been <u>fully executed</u> in E if $P_n = \epsilon$; otherwise P_0 is currently <u>executing</u> in E. In addition, intention P_0 in C_0 has been <u>successfully executed</u> in E if $P_i = nil$, for some $i \leq n$; intention P_0 has <u>failed</u> in E if it has been fully but not successfully executed in E.*

DEFINITION 4 (PROGRAM SIMULATION). *Let E be an execution of $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{P\} \rangle$. Program P' <u>simulates</u> program P <u>in execution</u> E iff there is an execution E' of configuration $C' = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{P'\} \rangle$ such that (a) $\overline{E}$ and $\overline{E'}$ are equivalent modulo P and P'; and (b) if P has been successfully executed in E, so has P' in E'. We say that P' <u>simulates</u> P iff P' simulates P in every execution of any configuration.*

We have, so far, defined the necessary technical foundations for adding HTN-style planning into the CAN BDI agent language, including substantially polishing and simplifying the original CAN's semantics from [24], incorporating extra representation for actions, and providing the necessary definitions of agent execution that were not addressed in [24]. Let us now move on to the core of the paper.

4. PLANNING IN BDI SYSTEMS

In this section, we shall integrate hierarchical planning into the BDI architecture of section 3. To do so, several issues need to

be addressed. Firstly, we want to keep the language as *uniform* as possible. Secondly, we allow control over when and on what planning is to be performed within the BDI architecture. Thirdly, we need to decide what domain information the planner will use—we want the planner to re-use as much information as possible from an existing BDI specification. Lastly, the result of the planning process ought to be carried on, and possibly monitored, within the BDI execution cycle in a uniform manner.

To address the above issues, we extend the CAN language by introducing a new language construct Plan for offline lookahead planning, so that $\mathsf{Plan}(P)$, where P is a plan-body program, is intended to mean "*plan for P offline, searching for a complete hierarchical decomposition.*" In this way, the BDI agent on Plan does a full lookahead search before committing to even the first step.

As with other constructs in the language, we need to provide the operational rules for the Plan construct. To do this, we shall distinguish, from now on, between two types of transitions on basic configurations, namely, bdi and plan (labelled) transitions. We write $C \xrightarrow{t} C'$ to specify a single step transition of type t (when no label is stated, both types apply). Intuitively, bdi-type steps will be used to model the normal BDI execution cycle, whereas plan-type transitions will be used to model (internal) deliberation steps within a *planning* context.

Following [6], the main operational rule states that configuration $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle$ can evolve to $\langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P') \rangle$ provided that $\langle \mathcal{B}, \mathcal{A}, P \rangle$ can evolve to $\langle \mathcal{B}', \mathcal{A}', P' \rangle$ from where it is possible to reach a *final* configuration in a finite number of *planning* steps:

$$\frac{\langle \mathcal{B}, \mathcal{A}, P \rangle \xrightarrow{\text{plan}} \langle \mathcal{B}', \mathcal{A}', P' \rangle \quad \langle \mathcal{B}', \mathcal{A}', P' \rangle \xrightarrow{\text{plan}_*} \langle \mathcal{B}'', \mathcal{A}'', nil \rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\text{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P') \rangle} \text{ Plan}$$

There are also two more simple rules associated with construct Plan that are shown in Figure 1. Rule Plan_t deals with the trivial case of planning on program *nil*; and, lastly, Plan_P handles the Plan construct within a planning context.

In addition to these three derivation rules for Plan, we need to restrict the two derivation rules $\rhd_f$ and G_R from section 3 to the bdi context only. This is because failure handling and goal restarting should not be made available during planning—they are features of the BDI execution cycle only. Hence, planning is *not* merely doing lookahead on the BDI execution cycle. We refer to the new versions of the rules as $\rhd_f^{bdi}$ and G_R^{bdi}, respectively. Also, since we now have two types of transition for basic configurations, we need to slightly modify the top-level agent rules A_{step} and A_{clean} to be defined in terms of bdi-type transitions. We only show here rules $\rhd_f^{bdi}$ and A_{step} (rules G_R^{bdi} and A_{clean} should be obvious):

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1 \rangle \xrightarrow{\text{bdi}}}{\langle \mathcal{B}, \mathcal{A}, (P_1 \rhd P_2) \rangle \xrightarrow{\text{bdi}} \langle \mathcal{B}, \mathcal{A}, P_2 \rangle} \rhd_f^{bdi}$$

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P \rangle \xrightarrow{\text{bdi}} \langle \mathcal{B}', \mathcal{A}', P' \rangle}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}', \mathcal{A}', (\Gamma \setminus \{P\}) \cup \{P'\} \rangle} A_{step}$$

Observe that, with the alternative rule $\rhd_f^{bdi}$, only the BDI execution cycle would be allowed to re-try alternative plans for an event upon the failure of some failed alternative. Indeed, a program of the form $(?false \rhd (\![\Delta]\!))$ has no transition within a plan context, whereas program $(\![\Delta]\!)$ would be tried within a bdi context.

In [24], it was required that the success and failure conditions in a goal-program be mutually exclusive. There is also another sensible restriction on goal-programs, namely, that the program P provided as a method for achieving a (declarative) goal ϕ_s does not make the failure condition ϕ_f true by itself.

DEFINITION 5. *A goal-program* $\mathsf{Goal}(\phi_s, P, \phi_f)$ *is* <u>coherent</u> *(relative to a plan library and an action library) if for every belief bases* $\mathcal{B}, \mathcal{B}', \mathcal{B}''$ *and sequences of actions* $\mathcal{A}, \mathcal{A}', \mathcal{A}''$ *such that* $\langle \mathcal{B}, \mathcal{A}, P \rangle \xrightarrow{\text{bdi}_*} \langle \mathcal{B}', \mathcal{A}', P' \rangle \xrightarrow{\text{bdi}_*} \langle \mathcal{B}'', \mathcal{A}'', nil \rangle$, *it is the case that* $\mathcal{B}' \not\models \phi_f$. *An agent is* <u>coherent</u> *if every goal-program mentioned in its plan library is coherent.*

From now on, we assume that agents are *coherent*—only the environment or other concurrent intentions may make the failure condition of a goal-program true.[4] As expected, if the agent's only intention $\mathsf{Plan}(P)$ is able to start executing, then there is at least one full successful BDI execution for such intention, provided there is no intervention from the outside environment. Equally important, under the same provisions, no execution of the agent will end up failing the intention.

THEOREM 1. *Let* $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \{\mathsf{Plan}(P)\} \rangle$ *such that* $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\text{bdi}}$. *If E is an environment-free agent execution of C, then intention $\mathsf{Plan}(P)$ is either executing or has been successfully executed in E. Moreover, there is an execution E^s of C in which intention $\mathsf{Plan}(P)$ has been successfully executed in E^s.*

PROOF. *This relies on the following lemma: if* $\langle \mathcal{B}, \mathcal{A}, P \rangle \xrightarrow{\text{plan}_*}$ $\langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$, *then* $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\text{bdi}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$.

On the contrary, suppose there is an environment-free execution E of the form $C_0 = C \cdot \ldots \cdot C_k$ *such that* $\langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k) \rangle \xrightarrow{\text{bdi}}\!\!\!\!\!/$. *Observe, though, that* $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\text{bdi}_k} \langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k) \rangle$. *By the rule* Plan, $\langle \mathcal{B}, \mathcal{A}, P \rangle \xrightarrow{\text{plan}_k} \langle \mathcal{B}_k, \mathcal{A}_k, P_k \rangle \xrightarrow{\text{plan}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$ *and* $\langle \mathcal{B}, \mathcal{A}, P \rangle \xrightarrow{\text{plan}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$ *applies. By using the above lemma, we get that* $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\text{bdi}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$. *Next, since* $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\text{bdi}_k}\!\!\!\!\!/ \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$, *there exist* $\mathcal{B}'', \mathcal{A}'', P''$ *such that* $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\text{bdi}} \langle \mathcal{B}'', \mathcal{A}'', P'' \rangle \xrightarrow{\text{bdi}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$. *Thus,* $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\text{bdi}}$ *and the execution E cannot exist.*

The second part follows easily from the fact that $\xrightarrow{\text{plan}_*}$ *stands for a finite chain of transitions: if the agent follows those exact transitions, P will eventually terminate successfully.* □

Thus, by using the new lookahead construct $\mathsf{Plan}(P)$, the programmer can make sure—to some extent—that *failing* executions of program P will be avoided. This contrasts with the usual (default) BDI execution of P which may potentially *fail* program P due to wrong decisions at choice points. Nonetheless, it should be clear that the proposed deliberation module is *local* in the sense that it does not take into account the potential interactions with the external environment and other concurrent intentions.

Let us now focus on the relationship between our planning construct Plan and existing HTN planners. To that end, we say that a CANPLAN agent is a <u>*bounded agent*</u> if its belief base and all belief conditions are defined in a language which follows the same constraints as those imposed by HTN planners [11] (e.g., first-order atoms, finite domains, close world assumption). It is worth pointing out that, in practice, most existing BDI programming language implementations do actualise such constraints and deal only with bounded agents. We also assume, without loss of generality, that bounded agents do not make use of $+b$ and $-b$ statements in their plans—only primitive actions can change the belief base. ($+b$ and $-b$ statements can always be represented using special actions.)

---

[4]This definition is a bit too strong in that it requires a goal-program to be "sound" w.r.t. the failure condition for *every* possible belief base and *every* chain of bdi-transitions, including failed recovered executions. Even though a weaker version could be obtained with a more involved definition, we stick, for simplicity, to the above one.

$$\frac{}{\langle\mathcal{B},\mathcal{A},(nil\triangleright P')\rangle \longrightarrow \langle\mathcal{B},\mathcal{A},nil\rangle}\ \triangleright_t \qquad \frac{}{\langle\mathcal{B},\mathcal{A},+b\rangle \longrightarrow \langle\mathcal{B}\cup\{b\},\mathcal{A},nil\rangle}\ +b \qquad \frac{}{\langle\mathcal{B},\mathcal{A},-b\rangle \longrightarrow \langle\mathcal{B}\setminus\{b\},\mathcal{A},nil\rangle}\ -b$$

$$\frac{\langle\mathcal{B},\mathcal{A},P_1\rangle \longrightarrow \langle\mathcal{B}',\mathcal{A}',P'\rangle}{\langle\mathcal{B},\mathcal{A},(P_1\triangleright P_2)\rangle \longrightarrow \langle\mathcal{B}',\mathcal{A}',(P'\triangleright P_2)\rangle}\ \triangleright \qquad \frac{}{\langle\mathcal{B},\mathcal{A},(nil\,;P)\rangle \longrightarrow \langle\mathcal{B},\mathcal{A},P\rangle}\ Seq_t \qquad \frac{}{\langle\mathcal{B},\mathcal{A},(P\parallel nil)\rangle \longrightarrow \langle\mathcal{B},\mathcal{A},P\rangle}\ \parallel_{t_2}$$

$$\frac{\langle\mathcal{B},\mathcal{A},P_1\rangle \longrightarrow \langle\mathcal{B}',\mathcal{A}',P'\rangle}{\langle\mathcal{B},\mathcal{A},(P_1\parallel P_2)\rangle \longrightarrow \langle\mathcal{B}',\mathcal{A}',(P'\parallel P_2)\rangle}\ \parallel_1 \qquad \frac{\langle\mathcal{B},\mathcal{A},P_2\rangle \longrightarrow \langle\mathcal{B}',\mathcal{A}',P'\rangle}{\langle\mathcal{B},\mathcal{A},(P_1\parallel P_2)\rangle \longrightarrow \langle\mathcal{B}',\mathcal{A}',(P_1\parallel P')\rangle}\ \parallel_2 \qquad \frac{}{\langle\mathcal{B},\mathcal{A},(nil\parallel P)\rangle \longrightarrow \langle\mathcal{B},\mathcal{A},P\rangle}\ \parallel_{t_1}$$

$$\frac{}{\langle\mathcal{B},\mathcal{A},\mathsf{Plan}(nil)\rangle \longrightarrow \langle\mathcal{B},\mathcal{A},nil\rangle}\ \mathsf{Plan}_t \qquad \frac{\langle\mathcal{B},\mathcal{A},P\rangle \xrightarrow{\text{plan}} \langle\mathcal{B}',\mathcal{A}',P'\rangle}{\langle\mathcal{B},\mathcal{A},\mathsf{Plan}(P)\rangle \xrightarrow{\text{plan}} \langle\mathcal{B}',\mathcal{A}',\mathsf{Plan}(P')\rangle}\ \mathsf{Plan_P}$$

$$\Delta = \{A_{step}, A_{clean}, A_{event}\} \cup \{Event, Sel, +b, -b, act, ?, Seq, Seq_t, \triangleright, \triangleright_t, \triangleright_f^{bdi}, \parallel_1, \parallel_2, \parallel_{t_1}, \parallel_{t_2}, \mathsf{G_I}, \mathsf{G}_s, \mathsf{G}_S, \mathsf{G}_R^{bdi}, \mathsf{Plan}, \mathsf{Plan}_t, \mathsf{Plan_P}\}.$$

Figure 1: CANPLAN**'s complete set of rules Δ is built from the rules described in the text plus the ones shown here.**

The next theorem establishes, formally, the link between the Plan construct and HTN planning. First, we prove that the new construct Plan could indeed be seen as an HTN planner. Second, we show that executions of program $\mathsf{Plan}(P)$ encode HTN plan solutions. Lastly, and not so surprisingly, we demonstrate that a straight-line HTN plan solution could be successfully executed by the BDI execution cycle. For clarity, we keep the translation between the BDI domain knowledge (i.e., libraries Π and Λ, and programs) and the HTN procedural knowledge (i.e., planning domain and task networks) implicit. (the theorem's proof is based on the relationship between the BDI's and HTN's entities as discussed in section 2.3.)

THEOREM 2. *For any bounded agent,*

1. $\langle\mathcal{B},\mathcal{A},\mathsf{Plan}(P)\rangle \xrightarrow{\text{bdi}}$ *iff* $sol(P,\mathcal{B},\Pi\cup\Lambda) \neq \emptyset$.

2. $\langle\mathcal{B},\mathcal{A},\mathsf{Plan}(P)\rangle \xrightarrow{\text{bdi}*} \langle\mathcal{B}',\mathcal{A}\cdot act_1\cdot\ldots\cdot act_k,\mathsf{Plan}(P')\rangle$ *with* $k\geq 1$ *iff there exists a plan* $\sigma\in sol(P,\mathcal{B},\Pi\cup\Lambda)$*, such that* $\sigma = act_1\cdot\ldots\cdot act_k\cdot\ldots\cdot act_n$*, for some* $n\geq k$.

3. *If there exists a plan* $\sigma = act_1\cdot\ldots\cdot act_n\in sol(P,\mathcal{B},\Pi\cup\Lambda)$*, then* $\langle\mathcal{B},\mathcal{A},(act_1;\ldots;act_n)\rangle \xrightarrow{\text{bdi}*} \langle\mathcal{B}',\mathcal{A}\cdot\sigma,nil\rangle$.

Therefore, provided we restrict to the language of HTN [11], our deliberator construct Plan provides a built-in HTN planner within the whole BDI framework. The above theorem is an important practical result as it gives us the rationale for using existing HTN planner systems, such as SHOP and SHOP2 within current BDI implementations, such as AGENTSPEAK and JACK.

4.1 Planning for Declarative Goals

So far we have seen how lookahead planning can be performed on (procedural) programs. Let us now discuss how (classical) planning for a *declarative* goal ϕ_s using a procedural program P can be done. There are a few choices for this and the following five properties that we may be interested in satisfying:

(A) P is used towards the eventual satisfaction of goal ϕ_s.
(B) P may execute partially if goal ϕ_s is achieved before P completion. That is, P need not be executed completely.
(C) There is a commitment to the goal ϕ_s so that P is reinstantiated and retried until the goal in question is established.
(D) There exists a mechanism for dropping the goal when a failure condition ϕ_f becomes true.
(E) At planning time, P is solved up to the point where the goal is met. That is, it may not be required to solve P completely.

The different alternatives that we shall consider, together with the properties satisfied by each one, are depicted in the following table:

ALTERNATIVES	A	B	C	D	E
$\mathsf{Plan}(P;?\phi_s)$	$\checkmark$				
$\mathsf{Plan}(\mathsf{Goal}(\phi_s,P,\phi_f))$	$\checkmark$	$\checkmark$		$\checkmark$	
$\mathsf{Goal}(\phi_s,\mathsf{Plan}(P),\phi_f)$		$\checkmark$	$\checkmark$	$\checkmark$	
$\mathsf{Goal}(\phi_s,\mathsf{Plan}(P;?\phi_s),\phi_f)$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	
$\mathsf{Goal}(\phi_s,\mathsf{Plan}(\mathsf{Goal}(\phi_s,P,\phi_f)),\phi_f)$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$

Interestingly, a *first-principles* account of planning can easily be obtained by using the first alternative $\mathsf{Plan}(P;?\phi)$ and taking $P = !seqActions$, where the distinguished event $seqActions$ can be solved with any sequence of primitive actions.

Notice that the last four alternatives make use of the special Goal construct available in CANPLAN to handle declarative goals within the BDI execution cycle. Observe also that the last option is the only one satisfying all five properties, combining then the advantages from the BDI execution cycle and the planning module. Consequently, it is sensible to define a new language construct $\mathsf{Plan}(\phi_s, P, \phi_f)$ that the user may use, as follows:

$$\mathsf{Plan}(\phi_s, P, \phi_f) \stackrel{\text{def}}{=} \mathsf{Goal}(\phi_s, \mathsf{Plan}(\mathsf{Goal}(\phi_s, P, \phi_f)), \phi_f).$$

Among other results, it can be shown that $\mathsf{Plan}(\phi_s, P, \phi_f)$ subsumes all the executions of $\mathsf{Plan}(\mathsf{Goal}(\phi_s, P, \phi_f))$.

THEOREM 3. *For every* ϕ_s, ϕ_f *and* P*, program* $\mathsf{Plan}(\phi_s, P, \phi_f)$ *simulates program* $\mathsf{Plan}(\mathsf{Goal}(\phi_s, P, \phi_f))$.

To recap: combinations of the Plan and Goal constructs suggest an interesting range of programs for declarative goals. We believe that $\mathsf{Plan}(\phi_s, P, \phi_f)$ provides a convenient mechanism for dealing with declarative goals at both planning and execution time.

4.2 Planning vs BDI Execution

We conclude this section by exploring the differences between the execution of a planning program and the normal BDI execution. A CANPLAN$^-$ agent is a CANPLAN agent whose plan language does not include the $\parallel$ and Goal constructs. This restriction corresponds to classical BDI agent programming languages like AGENTSPEAK and to *total-order* HTN planners like SHOP; neither system include concurrency and goals natively. Under such restricted CANPLAN agents, the planning module is no more than a lookahead mechanism on top of the BDI execution cycle.

THEOREM 4. *Program* P *simulates program* $\mathsf{Plan}(P)$ *in every* CANPLAN$^-$ *agent.*

On the other hand, when concurrency or goal-programs are considered, performing planning may result in extra executions. In fact, it can be shown that executing $\mathsf{Plan}(\mathsf{Plan}(P_1)\|P_2)$ is equivalent to executing $\mathsf{Plan}(P_1\|P_2)$, which in turn, is very different

from executing $(\mathsf{Plan}(P_1)\|P_2)$.[5] A similar situation arises with program $\mathsf{Plan}(\mathsf{Goal}(\phi_s, \mathsf{Plan}(P), \phi_f))$. The reason, informally, is that a Plan construct is ignored within the context of another Plan construct—there is no notion of planning within planning.

Surprisingly, also, the BDI execution engine may obtain successful executions that the planner cannot produce.

THEOREM 5. *There exists an agent configuration C of the form $\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{P\}\rangle$ for which there is an execution where P is successfully executed, but such that no execution of $C' = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{\mathsf{Plan}(P)\}\rangle$ can successfully execute $\mathsf{Plan}(P)$.*

PROOF. *Let us build a counter-example. Suppose that all actions are possible and that action act_1 just makes p true, that p and q are both false initially, and that there are only two plan rules in the plan library Π for handling event e: (i) $e : true \leftarrow act_1; ?q; act_2$; and (ii) $e : p \leftarrow act_3; act_2$. There is no solution for $\mathsf{Plan}(!e)$, but a BDI execution that would successfully execute $!e$ can be obtained by partially executing plan rule (i) (action act_1) and then, upon failure, fully executing plan rule (ii).* $\square$

As one can observe, the proof's counter-example relies on both the plan failure handling mechanism built into the BDI execution cycle and the programmer not having provided a full set of plans. In fact, if the plan library in the above proof's counter-example had included a third rule of the form $e : true \leftarrow act_1; ?p; act_3; act_2$, then the planner would have found a full execution. Still, as agent programs—that is, plan libraries—are often developed incrementally and in modules, the above situation could well arise.

It follows then that the combined framework of (default) BDI execution plus local hierarchical planning is strictly more general than hierarchical planning alone. Furthermore, as discussed after Theorem 1, by using the new local planning mechanism the programmer can rule out BDI executions that are bound to fail.

5. IMPLEMENTATION ISSUES

In earlier work [8], we presented an implementation that combined BDI reasoning with HTN planning. We used JACK[6] BDI system [3] and JSHOP[7] HTN planner, a Java version of SHOP [15]. Although the integrated framework does not fully realise the operational semantics presented here, it does incorporate some important concepts from it. In particular, it allows the programmer to specify from within a JACK program the points at which JSHOP should be called, in a manner similar to the Plan construct. Consistent with the semantics of Plan, JSHOP uses the same domain representation as JACK does (i.e., the plan library Π and belief base $\mathcal{B}$). In fact, the framework builds at runtime a JSHOP planning problem representation automatically from the JACK domain knowledge.

Some differences in the implementation arise from the nature of the systems chosen for the implementation. Since JSHOP is a total-order HTN planner, it cannot accommodate the concurrent construct $\|$. However, since parallelism has benefits, the integrated framework converts JSHOP's total-order solutions into partial-order solutions so that JACK can exploit possible parallelism at execution time. Some other differences exist between the implementation and the semantics for the sake of simplicity. For example, we excluded the $\mathsf{Goal}(\phi_s, P, \phi_f)$ construct in our system, as this construct does not have a direct matching concept in JACK or JSHOP. Including

this goal construct and using SHOP2 [16] to accommodate parallel execution of sub-goals natively are left for future work.

The main difference, however, is that the implementation *does not* re-plan at every step, as indicated by the Plan derivation rule defined in the semantics. This would clearly be unnecessarily inefficient. Instead, JSHOP was modified to return the relevant methods and bindings (rather than simply the actions); the BDI execution engine then follows step-by-step the decomposition suggested by the planner. Relevant environmental changes are detected by virtue of a step in the returned plan no longer being applicable within the BDI cycle. At that point, the planner is then called once again to provide an updated plan, and if none is available failure will occur in the BDI system. A disadvantage of this is that environmental changes leading to failure may be detected later in the implemented version than in the semantic rules. However, this drawback is offset by the much greater efficiency in what can be expected to be the majority of cases. This approach also has the benefit that an intention produced by a call to Plan will, in fact, *terminate*—successfully if there is no environmental interference. This is stronger than what Theorem 1 states, in which we needed to account for the strange, but potentially possible, situation where the Plan module continually returns a new and different plan prior to termination.

6. RELATED WORK

Except for INDIGOLOG [6], which is not *per se* a typical BDI agent programming language (see below), we are not aware of any other *formal* BDI-style agent programming language (e.g., 3APL [12], AGENTSPEAK [19], PRS [13]) providing a clean account of planning as we do here with CANPLAN. There are however a number of (implemented) *systems* or *frameworks* which do, in some way or another, mix planning and BDI-style execution. Some of these are *planners*, such as IPEM [1] and SAGE [14], that allow for the interleaving of action execution during the planning process. Others are *agent architectures*, such as RETSINA [17], CYPRESS and CPEF [23], and PROPICE-PLAN [10]; they are able to do lookahead planning. CYPRESS is based on the ACT [22] formalism that provides a uniform representation framework for BDI execution systems and hierarchical planners, hence supporting the type of mapping we have proposed in section 2.2. PROPICE-PLAN is perhaps the most similar system to ours, in that it is a typical BDI agent system that is able to call a planning module to find a solution for a particular problem. Like CANPLAN, a unified representation is used by both the planner and the BDI system. The work done in this paper differs, at least, in two ways form the above systems. First of all, we are particularly concerned with the *formal specification* of a BDI agent with built-in planning capabilities as well as with the *formal relation* between BDI systems and HTN planners. To our knowledge, none of the above systems come with a precise formal semantics. In some sense, however, our work was much motivated by the existence of these systems in an analogous way to how AGENTSPEAK [19] was motivated by systems like PRS [13]. Secondly, CANPLAN provides a mechanism for local deliberation *on-demand* that the programmer can use, as opposed to a fixed integration of planning within the execution engine (e.g., planning always [17] or just on (goal) failure [10, 23]).

Our work is possibly most related to that of De Giacomo and Levesque [6] in which INDIGOLOG, an incremental version of CONGOLOG [5] with a local deliberation module Σ, is proposed in the context of the situation calculus. Several ideas are taken from that work and applied to the BDI context. Our work is however different in that (i) INDIGOLOG is a cognitive agent language, with no explicit notions of events, plan library, plan selection, failure handling, intention base, etc., whereas our approach is linked to a

---

whole family of typical BDI languages and systems; (ii) our planning mechanism is provably linked to a well understood approach in the planning community, namely HTN planning, whereas, as far as we know, the INDIGOLOG deliberator module is very general and is not directly related to any planning system; and (iii) the integration of the planning module with the notion of declarative goals in CANPLAN has no counterpart in INDIGOLOG. In some ways, our approach has a more practical orientation than that of INDIGOLOG. It would be interesting to investigate the relations between INDIGOLOG and CANPLAN (e.g., identify the BDI subclass of agents that could be written and executed in INDIGOLOG).

7. CONCLUSION AND FUTURE WORK

We have proposed a mechanism for planning within BDI systems based on the intrinsic requirements of the BDI architecture. To do so, we provided an operational semantics that substantially simplifies and extends that presented in [24] to incorporate a new *planning* construct Plan. The new construct offers power and flexibility to the BDI programmer for specifying lookahead planning points in programs. We described results showing that the integration between the planning module and the whole BDI execution is the one intuitively expected, and proved that, under suitable assumptions, the planning task reduces to HTN planning. Lastly, we showed that the combined system allows a larger set of "good" executions than the planning module alone and discussed an implementation that incorporates many of the concepts from the semantics. We believe the work presented here is a significant step towards incorporating lookahead deliberation into BDI-type agent systems in a principled manner. More importantly, it provides a firm foundation for a range of interesting further work.

The fact that we have chosen to provide planning via a new construct is very much in the spirit of BDI systems, namely, allowing for direct encoding of programmer or domain expertise. In this case, the knowledge about when planning would be beneficial. It may be argued, though, that an intelligent agent should (also) make its own decisions as to when to plan. It is therefore worth investigating a more general account in which the agent could itself take the initiative to plan; for example, when all plans for a goal fail or when there is substantial spare time. *Re-planning* following failure of a plan produced by the planner module is also a topic we have not explored here and which deserves further work (see [23]).

We have already started exploring how to accommodate extensions to classical HTN planning within our formal framework. For instance, decoupling the hierarchical structure of BDI plans and using a planning account more akin to first principles would allow for potential discovery of new plan structures. This, in turn, could provide the basis for the agent to "learn" new plans that could be added to the plan library. Also, it can be useful to plan only to a certain level of abstraction or detail, leaving further remaining decompositions to execution time or until absolutely necessary as done in [4]. Both above generalisations are likely to require (or benefit from) extra representation of effects for high-level plans. Such extra representation would also provide support for reasoning about interactions of the plan being explored with other goals and intentions of the agent [4, 20]. In particular, we would like to extend our current *local* lookahead mechanism so that the agent considers, at least, all of its own active intentions when engaging in planning.

The framework presented here provides a basis for exploring the role of declarative goals [24, 21] in planning—preliminary results were given in section 4.1 but further investigation is needed.

Lastly, it would be interesting to develop *resource-bounded* version of our planning module. To that end, we are considering developing an *anytime* or *incremental* version of the Plan construct.

8. REFERENCES

[1] J. Ambros-Ingerson. *IPEM: Integrated Planning, Execution, and Monitoring*. PhD thesis, Dept. of Computer Science, University of Essex, U.K., 1987.

[2] M. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.

[3] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Components for Intelligent Agents in Java, AgentLink News Letter, Agent Oriented Software Pty. Ltd., Melbourne, January 1999.

[4] B. J. Clement and E. H. Durfee. Theory for Coordinating Concurrent Hierarchical Planning Agents Using Summary Information. In *Proc. of AAAI-99*, pages 495–502, 1999.

[5] G. De Giacomo, Y. Lespérance, and H. Levesque. ConGolog, A Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[6] G. De Giacomo and H. Levesque. An Incremental Interpreter for High-Level Programs with Sensing. In H. Levesque and F. Pirri, editors, *Logical Foundation for Cognitive Agents: contr. in honor of Ray Reiter*, pages 86–102. Springer, 1999.

[7] L. P. de Silva and L. Padgham. A Comparison of BDI Based Real-Time Reasoning and HTN Based Planning. In *Proc. of Australian Joint Conference on AI*, pages 1167–1173, 2004.

[8] L. P. de Silva and L. Padgham. Planning on Demand in BDI Systems. In *Proc. of ICAPS-05 (Poster)*, 2005.

[9] D. Dennett. *The Intentional Stance*. MIT Press, 1987.

[10] O. Despouys and F. F. Ingrand. Propice-Plan: Toward a Unified Framework for Planning and Execution. In *Proc. of European Conference on Planning*, pages 278–293, 1999.

[11] K. Erol, J. Hendler, and D. S. Nau. HTN Planning: Complexity and Expressivity. In *Proc. of AAAI-94*, pages 1123–1228, 1994.

[12] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[13] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44,1992.

[14] C. A. Knoblock. Planning, Executing, Sensing, and Replanning for Information Gathering. In *Proc. of IJCAI-95*, pages 1686–1693, 1995.

[15] D. S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In *Proc. of IJCAI-99*, pages 968–973, 1999.

[16] D. S. Nau, H. Muñoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. Total-Order Planning with Partially Ordered Subtasks. In *Proc. of IJCAI-01*, pages 425–430, 2001.

[17] P. Paolucci, O. Shehory, K. P. Sycara, K. P. Kalp, and A. Pannu. A Planning Component for RETSINA Agents. In *Proc. of ATAL-99*, pages 147–161, 1999.

[18] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI-FN-19, Dept. of Computer Science Department, Aarhus University, Denmark, 1981.

[19] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. V. Velde and J. W. Perram, editors, *Agents Breaking Away (LNAI)*, volume 1038 of *LNAI*, pages 42–55. Springer-Verlag, 1996.

[20] J. Thangarajah, L. Padgham, and M. Winikoff. Detecting & Exploiting Positive Goal Interaction in Intelligent Agents. In *Proc. of AAMAS-03*, pages 401–408, 2003.

[21] M. B. van Riemsdijk, M. Dastani, and J.-J. Ch. Meyer. Semantics of Declarative Goals in Agent Programming. In *Proc. of AAMAS-05*, pages 133–140, 2005.

[22] D. E. Wilkins and K. L. Myers. A Common Knowledge Representation for Plan Generation and Reactive Execution. *Journal of Logic and Computation*, 5(6):731–761, 1995.

[23] D. E. Wilkins and K. L. Myers. A Multiagent Planning Architecture. In *Proc. of AIPS-98*, pages 154–162, 1998.

[24] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & Procedural Goals in Intelligent Agent Systems. In *Proc. of KR-02*, pages 470–481, 2002.