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 behavior 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], ZAPL [12], JACK [3], CAN [24].

We would like to acknowledge the support of Agent Oriented Software and of the Australian Research Council under the grant “Learning and Planning in BDI Agents” (number LP0560702).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS’06 May 8–12 2006, Hakodate, Hokkaido, Japan.  
Copyright 2006 ACM 1-59593-303-4/06/0005 ...$5.00.

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-
traits, 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. We provide a brief abstract overview of these in order to comment on their similarities, as background for our integrated approach.

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 $B$, a set of recorded pending events (goals), a plan library $\Pi$, and an intention base $\Gamma$. The belief base encodes the agent’s knowledge about the world. The plan library contains plan rules of the form $e : \psi \rightarrow P$ encoding a plan-body or program $P$ for handling an event-goal $e$ when context condition $\psi$ 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, CAN [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 $act(\bar{x})$ that can be directly executed by the agent (e.g., $drive(x_1, x_2)$). A (high-level) compound task $c(\bar{x})$ is one that cannot be executed directly (e.g., $build_trip(origin, destin)$). A task network $d = [T, \phi]$ is a collection of tasks $T$ that need to be accomplished and a boolean formula of constraints $\phi$. Constraints impose restrictions on the ordering of the tasks ($e \prec e'$), on the binding of variables ($x \equiv x'$) 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 $(e, \psi, d)$ encodes a way of decomposing a high-level compound task $c$ into lower-level tasks using task network $d$ when $\psi$ holds. Methods provide the procedural knowledge of the domain. An HTN planning domain $D = (\Pi, \Lambda)$ consists of a library $\Pi$ of methods and a library $\Lambda$ of primitive tasks. Each primitive task in $\Lambda$ is a STRIPS style action with corresponding preconditions and effects in the form of add and delete lists. An HTN planning problem $P$ is the triple $(d, B, D)$ where $d$ is the task network to accomplish, $B$ is the initial belief state (i.e., a set of all ground atoms that are true in $B$), and $D$ is a planning domain. A plan $\sigma$ is a sequence $act_1, \ldots, act_n$ 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 $D$ 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 “backstracks” and tries an alternative reduction for a compound task previously reduced. If all compound tasks can eventually be reduced, a plan solution $\sigma$ is obtained.

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

$$sol_1(d, B, D) = comp(d, B, D),$$

$$sol_{n+1}(d, B, D) = sol_n(d, B, D) \cup \bigcup_{d' \in red(d, B, D)} sol_n(d', B, D).$$

Intuitively, $comp(d, B, D)$ is the set of all plan completion of a network $d$ containing only primitive tasks (i.e., plans for which the constraint formula $\phi$ in $d$ is satisfied), and $red(d, B, D)$ is the set of all reductions of $d$ in $B$ by methods in $D$. 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) 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.

<table>
<thead>
<tr>
<th>BDI SYSTEMS</th>
<th>HTN SYSTEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>belief base</td>
<td>state</td>
</tr>
<tr>
<td>plan library</td>
<td>method library</td>
</tr>
<tr>
<td>event</td>
<td>compound task</td>
</tr>
<tr>
<td>action</td>
<td>primitive task</td>
</tr>
<tr>
<td>plan-body/program</td>
<td>network task</td>
</tr>
<tr>
<td>plan rule</td>
<td>method</td>
</tr>
<tr>
<td>plan rule context</td>
<td>method precondition</td>
</tr>
<tr>
<td>test ?l in plan-body</td>
<td>state constraints</td>
</tr>
<tr>
<td>sequence in plan-body</td>
<td>ordering constraint $&lt;$</td>
</tr>
<tr>
<td>parallelism in plan-body</td>
<td>no ordering constraint</td>
</tr>
</tbody>
</table>
The above table is not complete—while 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 agents whereas the latter, in contrast, are concerned with hypothetical reasoning about actions and their potential interactions within a whole plan for achieving a goal/task.

3. THE BASIC BDI LANGUAGE

The CAN (Conceptual Agent Notation) 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 CONGLOG [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 (see [24]). 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 \( B \) and a set of plans \( P \). 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 \( \phi \)—a logical formula over the agent’s beliefs—follows from a belief set (i.e., \( B = \phi \)), and to add and delete a belief \( b \) to and from a belief base (i.e., \( B \cup \{b\} \) and \( B \setminus \{b\} \), 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 \( P \) consists of a collection of plan rules of the form \( e : \psi \leftarrow P \), where \( e \) is an event and \( \psi \) is the context condition which must be true in order for the plan-body \( P \) to be applicable. 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 \( \theta \), and events or (internal) achievement goals \( i \). Complex plans can be specified using sequencing \( P_1 \mid P_2 \), parallelism \( P_1 \parallel P_2 \), and declarative goals \( \text{Goal}(\phi_1, P, \phi_2) \) (explained later). Hence, the user language is described by the following grammar:

\[
P ::= act \mid +b \mid -b \mid ?\theta \mid \{\psi_1 \mid P_1 \mid P_2 \mid \text{Goal}(\phi_1, P, \phi_2)\}
\]

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 like \( P_1 \bowtie P_2 \), which executes \( P_1 \) and then executes \( P_2 \) only if \( P_1 \) failed, and \( \{\psi_1 \mid P_1 \mid \ldots \mid P_n \mid P_0 \} \), which is used to encode a set of (relevant) guarded plans. The full language is therefore described by the following grammar:

\[
P ::= nil \mid act \mid +b \mid -b \mid ?\theta \mid \{\psi_1 \mid P_1 \mid P_2 \mid \text{Goal}(\phi_1, P, \phi_2)\}
\]

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 \( T \) specifying the update semantics of basic actions is given: if \( T(\text{act}, B) \) is defined, it yields the new updated belief base \( B' \); otherwise, we say that the action’s precondition is not met in \( B \). However, for simplicity, we shall restrict ourselves to agents that are equipped with a (simple) STRIPS-like action description library \( \Lambda \) containing rules of the form \( \text{act} : \psi_\text{act} \leftarrow \Phi_\text{act} \); \( \text{act} \) one for each action type in the domain. Formula \( \psi_\text{act} \) corresponds to the action’s precondition, and \( \Phi_\text{act} \) and \( \Phi_\text{act} \) stand for the add and delete lists of actions, respectively. For example, action \( \text{move}(x, y, z) \), which moves object \( x \)

\[\text{move}(x, y, z) : \{\text{Free}(x, y) \land \text{At}(x, y) \rightarrow \{\text{Free}(y), \text{At}(x, y)\}; \{\text{Free}(y), \text{At}(x, z)\}\}\]

Next, we show the operational semantics for the above language along the lines of [24]. A transition relation \( \rightarrow \) on so-called configurations is defined by a set of derivation rules. A transition \( C \rightarrow C' \) specifies that executing configuration \( C \) a single step yields configuration \( C' \). We write \( C \rightarrow \) to state that there exists \( C' \) such that \( C \rightarrow C' \), and \( \rightarrow^* \) to denote the usual reflexive transitive closure of \( \rightarrow \). 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 basic configuration is a tuple \( (B, A, P) \) consisting of the current belief base \( B \) of the agent, the sequence \( A \) of primitive actions executed so far, and the plan-body \( P \) being executed (i.e., the current intention).

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

\[
\begin{align*}
\text{Event} & : (B, A, (A, \{\Delta\})) \xrightarrow{\psi_i : P_i \leftarrow \psi_i} (B, A, (\{\Delta\})) \\
\text{Seq} & : (B, A, (P_1 \parallel P_2)) \xrightarrow{\{\psi_1 \mid P_1 \mid P_2\}} (B, A, (P_1 \parallel P_2)) \\
\text{Act} & : (B, A, (\{P\})) \xrightarrow{a \cdot P \leftarrow \{\psi \mid \Delta\}} (B, A, (\{P\})) \\
\end{align*}
\]

Rule Event handles achievement goal events by collecting all relevant plans for the event in question. Rule Seq selects one applicable plan from a set of (remaining) relevant plans: program \( P \rightarrow \Delta \) states that program \( P \) should be tried first, falling back to the remaining alternatives in \( \Delta \) if required. Notice that plan rules’ context conditions are handled in a lazy manner. Rule Act 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 \( \Lambda \). Rule Seq handles sequencing of programs in the usual way. Rule \( \rightarrow^* \) is used along with rule Seq for failure handling: if the current plan \( P \theta \) for a goal fails (i.e., at some point the precondition of an action or a test goal is not met), rule \( \rightarrow^* \) applies first, and eventually, rule Seq may select another applicable alternative for the event-goal, if any.

A central distinguishing feature of CAN is its \( \text{Goal}(\phi_A, P, \phi_f) \) goal construct, which provides a mechanism for representing both declarative and procedural aspects of goals. Intuitively, a goal-program \( \text{Goal}(\phi_A, P, \phi_f) \) states that we should achieve the (declarative) goal \( \phi_A \) by using (procedural) plan \( P \); failing if \( \phi_f \) becomes

3Strictly speaking, the plan and action libraries \( \Lambda \) and \( \Delta \) may 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 \( \theta \) 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.
true. The execution of a goal-program is consistent with some desired properties of declarative goals (namely, persistent, possible, and unachieved). For instance, if \( P \) is fully executed but \( \phi_0 \) is still not true, \( P \) will be re-tried; and if \( \phi_0 \) becomes true during \( P \)'s execution, the whole goal will succeed immediately.

In order to capture the desired behavior 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:

\[
\begin{align*}
P \neq P_1 & \triangleright P_2 & \langle B, A, \text{Goal}(\phi, P, f) \rangle & \rightarrow \langle B, A, \text{Goal}(\phi, P \triangleright P, f) \rangle & \text{G}_f \\
B \models \neg \phi & \rightarrow \langle B, A, \text{nil} \rangle & \text{G}_a \\
B, A, \text{Goal}(\phi, P, f) & \rightarrow \langle B, A, \text{nil} \rangle & \text{G}_s \\
P \neq P_1 & \triangleright P_2 & \langle B, A, \text{Goal}(\phi, P, f) \rangle & \rightarrow \langle B, A, \text{Goal}(\phi, P \triangleright P, f) \rangle & \text{G}_g \\
B, A, \text{Goal}(\phi, P, f) & \rightarrow \langle B, A, \text{false} \rangle & \text{G}_f
\end{align*}
\]

When a goal-program is first encountered during execution, rule \( \text{G}_f \) "initializes" the execution of a goal-program by setting the program in the goal to \( P \triangleright P \), where the first \( P \) is to be executed and the second \( P \) is just used to carry the original plan \( P \) for (potential) use later on by rule \( \text{G}_f \). The second and third rules handle the cases where either the success condition \( \phi \) or the failure condition \( f \) becomes true. The fourth rule \( \text{G}_g \) is the one responsible for performing a single step on an already initialized goal-program. Notice that the second part in the pair \( P_1 \triangleright P_2 \) remains constant. Finally, rule \( \text{G}_a \) restarts the original program (stored as the second program in pair \( P_1 \triangleright P_2 \)) whenever the current program has finished, but the desired, and still possible, goal has not been achieved yet.

The above semantics for \( \text{Goal} \) 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 included also a goal base \( G \) 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/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 \( \langle N, \Lambda, \Pi, B, A, \Gamma \rangle \) where \( N \) is the agent name, \( \Lambda \) is an action description library, \( \Pi \) is a plan library, \( B \) is a belief base, \( A \) is the sequence of actions already performed by the agent, and \( \Gamma \) is the set of current intentions (i.e., plan-bodied). Transitions between agent configurations are dictated by the following three rules:

\[
\begin{align*}
P \in \Gamma & \quad \langle B, A, \Pi \rangle \rightarrow \langle B', A', \Pi' \rangle & \text{Astep} \\
& \quad \langle N, \Lambda, \Pi, B, A, \Gamma \rangle \leftarrow \langle N, \Lambda, \Pi, B', A', \Gamma' \setminus \{e\} \rangle & \text{Aevent} \\
P \in \Gamma & \quad \langle B, A, \Pi \rangle \rightarrow \langle N, \Lambda, \Pi, B, A, \Gamma \setminus \{P\} \rangle & \text{Aclean}
\end{align*}
\]

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 (i.e., 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 execution** \( E \) of an agent \( C_0 = \langle N, \Lambda, \Pi, B_0, A_0, \Gamma_0 \rangle \) is a possibly infinite sequence of agent configurations \( C_0, C_1, \ldots, C_n, \ldots \) such that \( C_i \Rightarrow C_{i+1} \) for every \( i \geq 0 \). A terminating execution is a finite execution \( C_0 \cdot \ldots \cdot C_n \) where \( C_n = \langle N, \Lambda, \Pi, B_n, A_n, \{\} \rangle \). An environment-free execution is one in which \( A_{\text{event}} \) 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—agents 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 **derived execution** \( \overline{E} \) is the sequence of configurations obtained from \( E \) by deleting all configurations \( C_j \) of the sequence such that \( B_j = B_{j+1} \) and \( A_j = A_{j+1} \).

In addition, we give the following notion 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 \( C_0, \ldots, C_n \) and \( C_0', \ldots, C'_n \) are equivalent modulo intentions if \( C_i = \langle N_i, \Lambda_i, \Pi_i, B_i, A_i, \Gamma_i \rangle \) for every \( 0 \leq i \leq n \). Also, the two executions are equivalent modulo intentions \( P_0 \in \Gamma_0 \) 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 \( C_0, C_1, \ldots, C_n \) for an agent \( C_0 = \langle N, \Lambda, \Pi, B_0, A_0, \Gamma_0 \rangle \), where \( \Gamma_0 = \Gamma_0' \cup \{P\} \). Intention \( P \) in \( C_0 \) has been fully executed in \( E \) if \( P_0 = \epsilon \); otherwise \( P_0 \) is currently executing in \( E \). In addition, intention \( P_0 \) in \( C_0 \) has been successfully executed in \( E \) if \( P_i = \epsilon \), for some \( i \leq n \); intention \( P_0 \) has failed in \( E \) if it has been fully but not successfully executed in \( E \).

**Definition 4 (Program Simulation).** Let \( E \) be an execution of \( C = \langle N, \Lambda, \Pi, B, A, \Gamma \cup \{P\} \rangle \). Program \( \text{P} \) simulates program \( \text{P} \) in execution \( E \) if there is an execution \( E' \) of configuration \( C' = \langle N, \Lambda, \Pi, B', A, \Gamma \cup \{P\} \rangle \) such that (a) \( E \) and \( E' \) are equivalent modulo program \( \text{P} \) and \( \text{P} ' \); and (b) if \( \text{P} \) has been successfully executed in \( E \), then \( \text{P} ' \) is in \( E' \). We say that \( \text{P} \) simulates \( \text{P} ' \) 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 Plan\(P\), where \(P\) is a plan-body, means “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 \(C \xrightarrow{\text{plan}} C'\) 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 \(\langle B, A, \text{Plan}(P)\rangle\) can evolve to \(\langle B', A', \text{Plan}(P')\rangle\) provided that \(\langle B, A, P\rangle\) can evolve to \(\langle B', A', P'\rangle\) from where it is possible to reach a final configuration in a finite number of planning steps:

\[
\langle B, A, \text{Plan}(P)\rangle \xrightarrow{\text{plan}} \langle B', A', \text{Plan}(P')\rangle \xrightarrow{\text{plan}} \langle B'', A'', \text{nil}\rangle
\]

There are also three extra simpler rules associated with construct Plan that are shown in Figure 1. Rule Plan will handle the case where no planning solution can be found; rule Plan deals with the trivial case of planning on nil; and, lastly, Plan 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 \(\Delta\) and \(G\) 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 \(\Delta_b\) and \(G_b\), respectively. Also, since we now have two types of transition for basic configurations, we need to slightly modify the top-level agent rules Astep and Aclean to be defined in terms of bdi-type transitions. We only show here rules \(\Delta_b\) and Astep (rules \(G_b\) and Aclean should be obvious):

\[
\langle B, A, P\rangle \xrightarrow{\Delta_b} \langle B', A, P'\rangle \xrightarrow{\text{bdi}} \langle B', A', P''\rangle
\]

\[
P \in \Gamma \Rightarrow \langle B, A, P\rangle \xrightarrow{\text{bdi}} \langle B', A', P'\rangle
\]

Observe that, with the alternative rule \(\Delta_b\), 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 \(\langle \text{false} \rangle \xrightarrow{\Delta}\) has no transition within a plan context, whereas program \(\langle \Delta\rangle\) 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 \(\phi\) does not make the failure condition \(\phi\) true by itself.

**Definition 5.** A goal-program \(\text{Goal}(\phi, P, \phi_f)\) is coherent (relative to a plan library and an action library) if for every belief base \(B, B', B''\) and sequences of actions \(A, A', A''\) such that \(\langle B, A, P\rangle \xrightarrow{\text{plan}} \langle B', A', P'\rangle \xrightarrow{\text{plan}} \langle B'', A'', \text{nil}\rangle\), it is the case that \(B' \neq \phi\). An agent is coherent 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.\(^{2}\) As expected, if the agent’s only intention 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 \(C = \langle \Pi, \Lambda, A, B, \{\text{Plan}(P)\}\rangle\) such that \(\langle B, A, \text{Plan}(P)\rangle \xrightarrow{\text{bdi}} \langle B', A', \text{Plan}(P')\rangle\). If \(E\) is an environment-free agent execution of \(C\), then intention Plan\(P\) is either executing or has been successfully executed in \(E\). Moreover, there is an execution \(E^*\) of \(C\) in which intention Plan\(P\) has been successfully executed in \(E^*\).

**Proof.** This relies on the following lemma: if \(\langle B, A, \text{Plan}(P)\rangle \xrightarrow{\text{plan}} \langle B', A', \text{Plan}(P)\rangle\), then \(\langle B, A, \text{Plan}(P)\rangle \xrightarrow{\text{bdi}} \langle B', A', \text{Plan}(P')\rangle\).

On the contrary, suppose there is an environment-free execution \(E\) of the form \(C_0 = C \cdot \ldots \cdot C_k\) such that \(\langle B_k, A_k, \text{Plan}(P_k)\rangle \xrightarrow{\text{bdi}} \langle B_k, A_k, \text{Plan}(P_k)\rangle\). Observe, though, that \(\langle B, A, \text{Plan}(P)\rangle \xrightarrow{\text{plan}} \langle B, A, \text{Plan}(P)\rangle\). By the rule Plan, \(\langle B, A, P\rangle \xrightarrow{\text{plan}} \langle B_k, A_k, P_k, \text{Plan}(P_k)\rangle\), and \(\langle B, A, P\rangle \xrightarrow{\text{plan}} \langle B', A', \text{Plan}(P')\rangle\). By the above lemma, we get that \(\langle B, A, \text{Plan}(P)\rangle \xrightarrow{\text{bdi}} \langle B', A', \text{Plan}(P)\rangle\). Next, since \(\langle B, A, \text{Plan}(P)\rangle \xrightarrow{\text{plan}} \langle B', A', \text{Plan}(P')\rangle\), there exist \(B'', A'', P''\) such that \(\langle B, A, \text{Plan}(P)\rangle \xrightarrow{\text{bdi}} \langle B', A', P''\rangle\). Thus, \(\langle B, A, \text{Plan}(P)\rangle \xrightarrow{\text{bdi}} \langle B', A', P''\rangle\) and the above \(E\) 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 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 bounded agent 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 \(+b\) states in their plans—only primitive actions can change the belief base. \(+b\) and \(+b\) states can always be represented via special BDI actions.

\(^2\)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.
The next theorem establishes, formally, the link between the Plan construct and HTN planning. First, we prove that the new construct Plan can indeed be seen as an HTN planner. Second, we show that executions of program 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 II and λ, and program) and the HTN procedural knowledge (i.e., planning domain, "task network" P) 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. \( \langle B, A, \text{Plan}(P) \rangle \vdash_b P \) iff \( \text{sol}(P; B, II \cup \Lambda) \neq \emptyset \).
2. \( \langle B, A, \text{Plan}(P) \rangle \vdash_b \langle B', A \cdot \text{act}_1 \ldots \text{act}_k, \text{Plan}(P') \rangle \) with \( k \geq 1 \) iff there exists a plan \( \sigma \in \text{sol}(P; B, II \cup \Lambda) \), such that \( \sigma = \text{act}_1 \ldots \text{act}_k \ldots \text{act}_n \), for some \( n \geq k \).
3. If there exists a plan \( \sigma = \text{act}_1 \ldots \text{act}_n \in \text{sol}(P; B, II \cup \Lambda) \), then \( \langle B, A, (\text{act}_1 \ldots \text{act}_n) \rangle \vdash_b \langle B', A \cdot \sigma, \text{nil} \rangle \).

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 [15] and SHOP2 [16], within current BDI implementations (e.g., AGENTSPEAK [19], JACK [3]).

4.1 Planning for Declarative Goals

So far we have seen how lookahead planning can be done on (procedural) programs. Let us now discuss how (classical) planning for a declarative goal \( \phi_s \), 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 \( \phi_s \).
(B) \( P \) may execute partially if goal \( \phi_s \) is achieved before \( P \) completion. That is, \( P \) need not be executed completely.
(C) There is a commitment to the goal \( \phi_s \), so that \( P \) is reinitiated and retried until the goal is established.
(D) There exists a mechanism for dropping the goal when a failure condition \( \phi_f \) 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 shown in the following table:

<table>
<thead>
<tr>
<th>ALTERNATIVES</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan(( P; ?\phi_s ))</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plan(( \text{Goal}(\phi_s, P, \phi_f) ))</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goal(( \phi_s, \text{Plan}(P), \phi_f ))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goal(( \phi_s, \text{Plan}(P; ?\phi_s), \phi_f ))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goal(( \phi_s, \text{Plan}(\text{Goal}(\phi_s, P, \phi_f)), \phi_f ))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Notice that the last four alternatives make use of the 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 Plan(\( \phi_s, P, \phi_f \)) in the following way:

\( \text{Plan}(\phi_s, P, \phi_f) \equiv \text{Goal}(\phi_s, \text{Plan}(\text{Goal}(\phi_s, P, \phi_f)), \phi_f) \).

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

THEOREM 3. For every \( \phi_s, \phi_f \) and \( P \), program Plan(\( \phi_s, P, \phi_f \)) simulates program Plan(\( \text{Goal}(\phi_s, P, \phi_f) \)).

To recap: combinations of the Plan and Goal constructs suggest an interesting range of programs for declarative goals. We believe that 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 CANPLANagent is a CANPLAN agent whose plan language does not include the \( \| \) 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 Plan(\( P \)) in every CANPLANagent.

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 Plan(\( \text{Plan}(P_1) \| P_2 \)) is equivalent to executing Plan(\( P_1 \| P_2 \)), which in turn, is very different
from executing \(\text{Plan}(P_1)\|P_2\). A similar situation arises with program \(\text{Plan}(\text{Goal}(\phi_1, \text{Plan}(P), \phi_2))\). The reason, technically, 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 \(<N, \Lambda, \Pi, B, A, \Gamma \cup \{P\}>\) for which there is an execution where \(P\) is successfully executed, but such that execution of \(C \prime = <N, \Lambda, \Pi, B, A, \Gamma \cup \{\text{Plan}(P)\}>\) can successfully execute Plan\((P)\).

**Proof.** Let us build a counter-example. Suppose that all actions are possible and that action \(act_2\) just makes \(p\) true, that \(p\) and \(q\) are both false initially, and that there are only two rule plans in the plan library \(\Pi\) 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 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).

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 \(e: true \leftarrow act_1; ?q; act_3; act_2\), then the planner would have found a full execution. Still, as agent programs 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 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 \(\Pi\) and belief base \(B\)). 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 does not use the \(\parallel\) construct defined in \(P\). 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 semantics for the sake of simplicity. For example, we exclude the Goal\((\phi_1, P, \phi_2)\) construct in our system, as this construct does not have a direct matching concept in JACK or JSHOP. Including this goes beyond what we have proposed in previous work [8, 9].

---

\(^6\)A framework where \(\text{Plan}(\text{Plan}(P_1)\|P_2)\) is not equivalent to \(\text{Plan}(P_1\|P_2)\) would require an account of HTN planning within an HTN planner. This framework can be obtained by dropping rule Plan\(_{\parallel}\) and making rule Plan also available within the plan context.

\(^7\)www.agent-software.com.au

---

**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., AGENTS-PEAK [19], 3APL [12], 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 IPREM [1] and SAGE [14], that allow for the interleaving of action execution during the planning process. Others are agent architectures, such as RETSINA [17], CYCRESS and CPEF [23], and PROPICE-PLAN [10]; they are able to do lookahead planning. CYCRESS 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 \(\Sigma\), 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 ICILOGOLOG delimiter module is very general and is not directly related to any planning system; (iii) the integration of the planning module with the notion of declarative goals in CANPLAN has no counterpart in ICILOGOLOG. In some ways, our approach has a more practical orientation than that of ICILOGOLOG. It would be interesting to investigate the relations between ICILOGOLOG and CANPLAN (e.g., identify the BDI subclass of agents that could be written and executed in ICILOGOLOG).

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 performing planning.

The framework presented here provides a basis for exploring the interaction between declarative goals [24, 21]—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

B. Auxiliary Definitions

Definition 6 (Remove Plan Transformation RP(P₁, P₂) and RPA(P₁)). Let RP(P₁, P₂) be a relation that holds iff P₂ is exactly like P₁ with 0 ≤ m ≤ n occurrences of the Plan construct in P₁, removed, where n is the number of occurrences of the Plan constructs in P₂. Finally, let RPA(P₁) be the program obtained from deleting all Plan constructs in P₁ (clearly RPA(P₁) = RP(P₁, P₂)).

B.1 Execution Simulation

Definition 7 (Execution Simulation). Let C and C’ be two configurations, probably for different agents. Let E be an execution of C. Configuration C’ simulates configuration C in execution E iff there is an execution E’ of C’ such that (a) E and E’ are equivalent, and (b) if E is terminating, so is E’. Configuration C’ simulates C in every execution of C.

Definition 8 (Derived Transitions). An internal step transition →_i is obtained by restricting →: R → R’ iff R → R’ and O(R) = O(R’). The visible transition relation →_v is defined by: R → R’ iff there exists R” such that R →_v R” and R” → R’ but R” →_v R’ (i.e., O(R”) = O(R’)).

Definition 9 (Simulation Relation). A binary relation S = (B₁, B₂) between programs is a simulation relation w.r.t. transition → iff for every P₁ and P₂ such that S(P₁, P₂), it is the case that for every B, A:

1. if (B, A, P₁) →_i (B’, A’, nil), then (B, A, P₂) →_i (B’, A’, nil).
2. if (B, A, P₁) → (B’, A’, P₂), there exists R’, R”, P’ such that (a) (B, A, P₁) →_i R’, R” → R’ or R” = R’ and R” →_v (B’, A’, P’); and (b) S = (P’₁, P’₂).

C. Auxiliary Results

The following Lemma states that btd-step transitions completely ignore the second plan in programs of the form P₁ ⊳ P₂.

Lemma 6 (Properties of ⊳ in plan-type transitions). For every B, A and P₁, P₂:

1. (B, A, P₁ ⊳ P₂) plan (B’, A’, nil) iff (B, A, P₁) plan (B’, A’, nil);
2. If (B, A, P₁) plan, then (B, A, fail ⊳ P₂) plan;
3. (B, A, Goal(φ₁, P₁ ⊳ P₂, φ₂)) plan (B’, A’, nil) iff (B, A, Goal(φ₁, P₁, φ₂)) plan (B’, A’, nil);
4. If (B, A, P₁) plan and B ↑ φ₁ ∨ φ₂, then (B, A, Goal(φ₁, P₁ ⊳ P₂, φ₂)) plan.

Proof. Direct from the fact that rules ⊳ and Goal are not available within the plan context.

Lemma 7. If (B, A, P) plan (B’, A’, P’), then (B, A, P) btd (B’, A’, P’).

Proof. Direct from the fact that any plan-type derivation rule except for Plan, is also available as a btd-type.

Lemma 8. Let P₁ and P₂ = RPA(P₁). Then, (B, A, Plan(P₁)) btd (B’, A’, P’₁) iff A’ = A and P₂ = RPA(P’₁) or (ii) (B, A, Plan(P₂)) btd (B’, A’, Plan(RPA(P’₁))).

Proof. By assumption we know that (a) (B, A, P₁) plan (B’, A’, P’₁), and (b) (B’, A’, P’₁) plan (B’, A, nil). Let us consider the following two cases on (a). If transition (a) is supported by derivation rule Plan, then clearly B’ = B, A’ = A and RPA(P’₁) (that is, some subprogram Plan(nil) in P₁ was removed to just nil). In this case, it is obvious that P₂ = RPA(P’₁) and (i) applies. Now suppose transition (a) is not supported by rule Plan, and let us take P’₂ = RPA(P’₁). Because any Plan construct in P₁ is completely ignored due to rule Plan, it follows that (B, A, Plan(P₂)) btd (B’, A’, Plan(P’₂)).

C.1 Properties of Plan

The following Lemma states that the planning module on a program of the form P₁ ⊳ P₂ is fully committed to the first program P₁ only.


Proof. This is a direct consequence of point (1) in Lemma 6.

Theorem 10 (Nested Plan). The following are some auxiliary properties of Plan:

1. (B, A, Plan(Plan(P’))) btd (B’, A’, Plan(Plan(P’))), if (B, A, Plan(P’) btd (B’, A’, P’)).
2. (B, A, Plan(Plan(P’))) btd (B’, A’, nil) if (B, A, Plan(P’) btd (B’, A’, nil)).

Proof. For the first part we have:

(B, A, Plan(Plan(P’))) btd (B’, A’, Plan(Plan(P’)))

(B, A, Plan(P’)) plan (B’, A’, Plan(P’)) and (B’, A’, Plan(P’)) plan (B’, A’, nil)

(B, A, P) plan (B’, A’, P’) and (B’, A’, P’) plan (B’, A’, nil)

(B, A, Plan(P’)) btd (B’, A’, Plan(P’))

For the second part we have:

(B, A, Plan(Plan(P’))) btd (B, A, Plan(nil)) if (B, A, Plan(P’)) plan (B’, A’, nil) if P = nil if (B, A, Plan(P’) btd (B, A, nil).

□
C.1.1 Interaction between Plan and Goal

**Theorem 11.** Let \( R = (\mathcal{B}, \mathcal{A}, \text{Plan}(\text{Goal}(\phi_s, P, \phi_f))) \) such that \( \mathcal{B} \models \phi_s \land \phi_f \). Then,

- if \( \mathcal{B} \models \phi_s \), then there exists \( R' = (\mathcal{B}, \mathcal{A}, \text{Plan}(\text{nil})) \) such that \( R \xrightarrow{\text{ndi}} R' \), and for every \( R'' \) such that \( R \xrightarrow{\text{ndi}} R'' \), \( R'' = R' \).
- if \( \mathcal{B} \models \phi_f \), then \( R \xrightarrow{\text{ndi}} \).

**Proof.** If \( \mathcal{B} \models \phi_s \), then the only applicable derivation rule is \( \text{G}_s \) and \( R' = (\mathcal{B}, \mathcal{A}, \text{Plan}(\text{nil})) \). Similarly, if \( \mathcal{B} \models \phi_f \), then for every \( R' \) such that \( \mathcal{B}, \mathcal{A}, \text{Goal}(\phi_s, P, \phi_f) \) \( \xrightarrow{\text{plan}} \) \( R' \), it is the case that \( R' = (\mathcal{B}, \mathcal{A}, \text{false}) \).

**Theorem 12.** (Properties of Program Plan(\( \phi_s, P, \phi_f \))) Let \( C_0 = \langle \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma_0 \cup \{ P_0 \} \rangle \) be an agent configuration with \( P_0 = \text{Plan}(\phi_s, P, \phi_f) \) (\( P \) is a user program). Let \( E = C_0 \cdot \ldots \cdot C_k \) be an execution where \( P_0 \) is currently executing and let \( P^\# = \text{Plan}(\text{Goal}(\phi_s, P, \phi_f)) \). Then, \( P_k \in \{ P_0, \text{Goal}(\phi_s, \text{Plan}(\text{Goal}(\phi_s, P^\# \triangleright P, \phi_f)) \triangleright P^\#, \phi_f), \text{nil}, \text{false} \} \).

If \( P_k^b = \text{Goal}(\phi_s, \text{Plan}(\text{Goal}(\phi_s, P^\# \triangleright P, \phi_f)) \triangleright P^\#, \phi_f) \), then \( (\mathcal{B}_k, \mathcal{A}_k, P_k^b) \xrightarrow{\text{ndi}} C_k^b \) \( \Rightarrow \) and the following hold:

Suppose \( (\mathcal{B}_k, \mathcal{A}_k, P_k^b) \xrightarrow{\text{ndi}} R \). If \( \mathcal{B} \models \phi_s \), then \( R = (\mathcal{B}_k, \mathcal{A}_k, \text{nil}) \). If \( \mathcal{B} \models \phi_f \), then \( R = (\mathcal{B}_k, \mathcal{A}_k, \text{false}) \).

Suppose \( \mathcal{B} \not= \phi_s \lor \phi_f \). Then:

\[
\langle \mathcal{B}_k, \mathcal{A}_k, P_k^b \rangle \xrightarrow{\text{plan}} (\mathcal{B}', \mathcal{A}', P') \xrightarrow{\text{plan}} (\mathcal{B}'', \mathcal{A}'', P'') \quad \text{iff} \quad \mathcal{B}' \models \phi_s.
\]

\[
\langle \mathcal{B}_k, \mathcal{A}_k, P_k^b \rangle \xrightarrow{\text{plan}} (\mathcal{B}', \mathcal{A}', \text{Goal}(\phi_s, \text{Plan}(\text{Goal}(\phi_s, P' \triangleright P, \phi_f)) \triangleright P^\#, \phi_f))
\]

there is no \( R \) such that

\[
\langle \mathcal{B}_k, \mathcal{A}_k, P_k^b \rangle \xrightarrow{\text{plan}} R \xrightarrow{\text{plan}} (\mathcal{B}'', \mathcal{A}'', P''), \mathcal{B}'' \models \phi_s
\]

\[
\langle \mathcal{B}_k, \mathcal{A}_k, P_k^b \rangle \xrightarrow{\text{plan}} (\mathcal{B}_k, \mathcal{A}_k, \text{Goal}(\phi_s, P^\# \triangleright P^\#, \phi_f))
\]
D. PROOF OF THEOREM 1

**Lemma 13.** For every $B, A$ and $P$, if $\langle B, A, P \rangle \xrightarrow{\text{plan}} \langle B, A, \text{Plan}(P) \rangle$, then $\langle B, A, \text{Plan}(P) \rangle \xrightarrow{\text{bdi}} \langle B_f, A_f, \text{nil} \rangle$.

**Proof.** We prove this by induction on the length $n$ of the plan-type derivation.

**Base case:** Suppose $n = 0$. Then $P = \text{nil}$, $B_f = B$, and $A_f = A$. By using derivation rule $\text{Plan}$, we get $\langle B, A, \text{Plan}(P) \rangle \xrightarrow{\text{bdi}} \langle B_f, A_f, \text{nil} \rangle$.

**Inductive Case:** Suppose $n = k + 1$. Then, there exists $R^* = \langle B^*, A^*, P^* \rangle$ such that (a) $\langle B, A, P \rangle \xrightarrow{\text{plan}} R^*$ and (b) $R^* \xrightarrow{\text{plan}} \langle B_f, A_f, \text{nil} \rangle$.

Using (a) and (b), we can use derivation rule $\text{Plan}$ to obtain $\langle B, A, \text{Plan}(P) \rangle \xrightarrow{\text{bdi}} \langle B^*, A^*, \text{Plan}(P^*) \rangle$. Moreover, by (b) and the induction hypothesis, $\langle B^*, A^*, \text{Plan}(P^*) \rangle \xrightarrow{\text{bdi}} \langle B_f, A_f, \text{nil} \rangle$. Thus, $\langle B, A, \text{Plan}(P) \rangle \xrightarrow{\text{bdi}} \langle B_f, A_f, \text{nil} \rangle$ follows.

Using the above auxiliary lemma, we now prove the main result.

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

**First Claim:** On the contrary, assume that $\text{Plan}(P)$ failed in some execution. Then, there is an environment-free execution $E$ of the form $C_0 = C, C_1, \ldots, C_k$, such that $k \geq 1$, $C_k = \langle \mathcal{N}, \Lambda, \Pi, B_k, A_k, \{\text{Plan}(P_k)\} \rangle$, and $\langle B_k, A_k, \text{Plan}(P_k) \rangle \xrightarrow{\text{bdi}}$. That is, the execution of $C$ ended up in configuration $C_k$ where the original intention $\text{Plan}(P)$ is stuck.

We know that $\langle B_{k-1}, A_{k-1}, \text{Plan}(P_{k-1}) \rangle \xrightarrow{\text{bdi}} \langle B_k, A_k, \text{Plan}(P_k) \rangle$ which means that (a) $\langle B_{k-1}, A_{k-1}, P_{k-1} \rangle \xrightarrow{\text{plan}} \langle B_k, A_k, P_k \rangle$; and (b) there exists $B_f, A_f$ such that $\langle B_k, A_k, P_k \rangle \xrightarrow{\text{plan}} \langle B_f, A_f, \text{nil} \rangle$. By Lemma 13, $\langle B_k, A_k, \text{Plan}(P_k) \rangle \xrightarrow{\text{bdi}} \langle B_f, A_f, \text{nil} \rangle$. Because $\text{Plan}(P_k) \neq \text{nil}$, $\langle B_k, A_k, \text{Plan}(P_k) \rangle$ follows (i.e., there is at least one possible next transition).

**Second Claim:** By assumption, there exist $B_f$ and $A_f$ such that $\langle B, A, P \rangle \xrightarrow{\text{plan}^*} \langle B_f, A_f, \text{nil} \rangle$ hold. By Lemma 13, we know that $\langle B, A, \text{Plan}(P) \rangle \xrightarrow{\text{bdi}} \langle B_f, A_f, \text{nil} \rangle$. It is not hard to see that we can use this basic $\text{bdi}$-type derivation to construct an environment-free agent execution $E_\omega$ for $C$ such that the original intention $\text{Plan}(P)$ is successfully executed in it.

An immediate consequence of this theorem is the following corollary.

**Corollary 14.** Let $E$ be a terminating environment-free execution of agent $\langle \mathcal{N}, \Lambda, \Pi, B, A, \{\text{Plan}(P)\} \rangle$. Furthermore, suppose that $\langle B, A, \text{Plan}(P) \rangle \xrightarrow{\text{bdi}}$. Then, intention $\text{Plan}(P)$ has been successfully executed in $E$.

**Proof.** From Theorem 1 and the fact that if $E$ is terminating, then it is the case that intention $\text{Plan}(P)$ has either been successfully executed in $E$ or it has failed in $E$. 
E. TRANSLATION OF BDI LIBRARIES TO HTN DOMAIN KNOWLEDGE: THEOREM 2

Here, we show how to translate a CANPLAN plan library \( \Pi \) and action description library \( \Lambda \) into a planning domain \( \mathcal{D} = (Op(\Lambda), Me(\Pi)) \).

**Definition 10 (Bounded BDI agents).** We say that a CANPLAN agent is a bounded agent if its belief base and all belief conditions are defined in a language which follows the same constraints as those imposed by HTN planners \([1] \) (e.g., first-order atoms, finite domains, close world assumption). Moreover, a plan-body in a bounded agent cannot include explicit addition or deletion of belief statements (i.e., \(+b \) and \(-b \) statements).

From now on, we assume that CANPLAN agents are bounded. In what follows, we will show two theorems which are specific version of Theorem 2. Theorem 15 is the most trivial version where the translation from the BDI language to the HTN one is almost trivial. However, such theorem holds only for agents that do not make use of the Goal construct. Theorem 17 extends Theorem 15 to accommodate Goal-programs and relies on a complex transformation for such specific programs. Finally, we discuss how such transformation can be avoided all together if we slightly change the semantics of HTN planning.

### E.1 Converting BDI Belief Conditions into HTN Constraints

Given a formula \( \phi \), we define \( \phi^* \) and \( (\phi, n)^* \) inductively as follows:

1. If \( \phi = l \), then \( \phi^* = l \) and \( (\phi, n)^* = (l, n) \).
2. If \( \phi = l_1 \land l_2 \), then \( \phi^* = l_1 \land l_2 \) and \( (\phi, n)^* = (l_1, n) \land (l_2, n) \).
3. If \( \phi = l_1 \lor l_2 \), then \( \phi^* = l_1 \lor l_2 \) and \( (\phi, n)^* = (l_1, n) \lor (l_2, n) \).
4. If \( \phi = \neg \phi_1 \), then \( \phi^* = \neg \phi_1 \) and \( (\phi, n)^* = (\phi_1, n)^* \).
5. If \( \phi = (\phi_1 \land \phi_2) \), then \( \phi^* = (\phi_1 \land \phi_2) \) and \( (\phi, n)^* = (\phi_1, n)^* \land (\phi_2, n)^* \).
6. If \( \phi = (\neg \phi_1 \lor \phi_2) \), then \( \phi^* = (\neg \phi_1 \lor \phi_2) \) and \( (\phi, n)^* = (\neg \phi_1 \lor \phi_2, n)^* \).

The definitions of \( (n, \phi)^* \) and \( (n_1, \phi, n_2)^* \) are analogous.

### E.2 Converting BDI Action Description Libraries into HTN Operators: \( Op(\Lambda) \)

Suppose that the (bounded) CANPLAN agent contains an action description library \( \Lambda \) with actions of the following form:

\[
act(\vec{x}) : l(\vec{x}) \land \ldots \land l_n(\vec{x}) \leftrightarrow \{a^1_1(\vec{x}), \ldots, a^1_m(\vec{x})\}^C \wedge \{a^1_1(\vec{x}), \ldots, a^1_m(\vec{x})\}^+,
\]

where \( l_i \) are literals, and \( a^1_i \), \( a^+_i \) are atoms. Given a library of actions \( \Lambda \), we define the corresponding set of HTN primitive tasks/operators \( Op(\Lambda) \) as follows:

\[
Op(\Lambda) = \{(act(\vec{x}), \{\text{pre} : l(\vec{x}), l_n(\vec{x})\}, \{\text{post} : a^-_1(\vec{x}), \ldots, a^-_m(\vec{x}), a^+_1(\vec{x}), \ldots, a^+_m(\vec{x})\}) \mid \act(\vec{x}) : l(\vec{x}) \land \ldots \land l_n(\vec{x}) \leftrightarrow \{a^-_1(\vec{x}), \ldots, a^-_m(\vec{x})\}^C \wedge \{a^+_1(\vec{x}), \ldots, a^+_m(\vec{x})\}^+ \in \Lambda\}.
\]

### E.3 Converting Goal-free BDI Plan Libraries into HTN Method Libraries: \( Me(\Pi) \)

In this section, we show how a BDI plan library \( \Pi \) that contains no Goal-program can be mapped into an HTN method library \( Me(\Pi) \). We shall later consider libraries that make use of Goal-constructs. In the simple case, the mapping is quite straightforward.

Below, \( \{P(\vec{x})\} \) refers to the size of program \( P(\vec{x}) \) measured as the number of complex constructs in it. We shall also use a special operator (or primitive task) \texttt{dummyTask}, which is always possible and has no effects whatsoever when executed. When \( \sigma \) is an HTN plan, \texttt{clean}(\sigma) stands for plan \( \sigma \) with all actions “\texttt{dummyTask}” operators (primitive tasks) deleted.

Given a bounded BDI CANPLAN library \( \Pi \) mentioning no goal-programs, we define its corresponding HTN method-library \( Me(\Pi) \) as follows:

\[
Me(\Pi) = \bigcup_{\langle e(\vec{x}), (\vec{y}, g) \rightarrow P(\vec{x}, \vec{y}, g)\rangle \in \Pi} \{\langle e(\vec{x}), \psi(\vec{x}, g), T_1(P(\vec{x}, \vec{y}, \vec{z}, 0)), T_2(P(\vec{x}, \vec{y}, \vec{z}, 0))\rangle \}
\]

Function \( T(P, n) \) maps a plan-body \( P \) and a natural number \( n \) into a pair \( (T, M) \), where \( T \) is an HTN task-network and \( M \) is a set of methods. We write \( T_1(P, n) \) to refer to the first argument \( T \) of the pair, and \( T_2(P, n) \) to refer to the second argument \( M \)—that is, if \( T(P, n) = (T, M) \) then \( T_1(P, n) = T \) and \( T_2(P, n) = M \). Function \( T(P, n) \) is defined inductively on the structure of \( P \) as follows:

**Base Cases:** In this case, the program can be a primitive action, a test condition, an event, or the trivial program \texttt{nil}:

- If \( P(\vec{x}) = \text{act}(\vec{x}) \), then \( T(P(\vec{x}), n) = \{(n : \text{act}(\vec{x})), \text{nil}, \emptyset\} \).
- If \( P(\vec{x}) = \text{?}(\phi(\vec{x})) \), then \( T(P(\vec{x}), n) = \{(n : \text{dummyTask}, (\phi(\vec{x}), n)^*), \emptyset\} \).
- If \( P(\vec{x}) = e(\vec{x}) \), then \( T(P(\vec{x}), n) = \{(n : e(\vec{x})), \text{nil}, \emptyset\} \).
- If \( P(\vec{x}) = \text{nil} \), then \( T(P(\vec{x}), n) = \{(n : \text{dummyTask}, \text{nil}), \emptyset\} \).

**Inductive Cases:** Suppose \( P_0(\vec{x}) \) is any of the programs in the above base cases, that is, \( P_0(\vec{x}) = \text{act}(\vec{x}) \mid \text{?}(\phi(\vec{x})) \mid \text{nil} \mid e(\vec{x}) \). Then,

- If \( P(\vec{x}) = (P_0(\vec{x}) ; P^+(\vec{x})) \), then \( T(P(\vec{x}), n) = \{(T_0 \cup T', (n < n + 1) \land C_0 \land C') \}, \emptyset\) \),

- If \( P(\vec{x}) = (P_0(\vec{x}) \mid P_2(\vec{x})) \), then \( T(P(\vec{x}), n) = \{(T_1 \cup T_2, C_1 \land C_2) \}, \emptyset\) \).
- If \( P(\vec{x}) = (P_0(\vec{x}) \triangleright P_2(\vec{x})) \), then \( T(P(\vec{x}), n) = \{(P_1(\vec{x}), n) \} \).
- If \( P(\vec{x}) = \psi(\vec{x}) : P_0(\vec{x}) \), then \( T(P(\vec{x}), n) = \{(n : \text{choice}^P(\vec{x}), \text{nil}), \bigcup_{i=1}^k \{(\text{choice}^P(\vec{x}), \psi(\vec{x})^*, T_1(P_i(\vec{x}), 0)) \} \cup T_2(P_1(\vec{x}), 0)\} \).
E.4 Converting Full BDI Plan Libraries into BDI Method Libraries: $M^e_+(\Pi)$

We shall now consider BDI plan libraries and plan-bodies that may mention Goal-programs. To that end, we will extend the transformation given in Appendix E.3 to include Goal-programs; this transformation is a bit more involved given that HTN does not accommodate naturally any construct of that sort. We start by converting a BDI plan library into one that is Goal-free.

E.4.1 Mapping Goal-programs to Goal-free

Now, let us consider goal-programs of the form $\text{Goal}(\phi_s, P, \phi_f)$. Informally, we will construct a library $\Pi^*$ from an original library $\Pi$ in such a way that executing goal-program $\text{Goal}(\phi_s, P, \phi_f) \mid w.r.t. library \Pi$ is equivalent to executing program $P \mid w.r.t. library \Pi^*$. We observe this transformation does not comply with the notion of eliminability (see [7]) as it substantially changes the structure of the agent.

**Definition 11** (Program and Library Transformations $P^*$ and $\Pi^*$). Let $P$ be a plan-body and let $\gamma(\overline{w})$ be a belief condition. The plan body $P^*$ is the plan-body obtained from $P$ as follows:

1. Every event $e(\overline{t})$ mentioned in $P$ is replaced with the (new) event $e'(\overline{t}, \overline{w})$ in $P^*$.
2. Every test condition $\phi(\overline{t})$ mentioned in $P$ is replaced with the test condition $\phi'(\overline{t}) \cup \gamma(\overline{w})$ in $P^*$.
3. Every action $act(\overline{t})$ mentioned in $P$ is replaced with the (new) event $\gamma_{\gamma_{\phi}}'(\overline{t}, \overline{w})$ in $P^*$.

When $\Pi$ is a plan library, we build the new library $\Pi^*$ from $\Pi$, as follows:

1. Every plan-rule of the form $e(\overline{x}) : \psi(\overline{x}, \overline{y}) \leftarrow P(\overline{x}, \overline{y}, \overline{z})$ is replaced with the following two plan-rules:

   \[
   e'(\overline{x}, \overline{w}) : \neg\gamma(\overline{w}) \land \psi(\overline{x}, \overline{y}) \leftarrow P^*,
   e'(\overline{x}, \overline{w}) : \gamma(\overline{w}) \leftarrow nil.
   \]

2. For every action $act(\overline{x})$ in the domain, the following two plan-rules are included:

   \[
   e_{\gamma_{\phi}}^{\gamma_{\phi}}(\overline{x}, \overline{w}) : \neg\gamma(\overline{w}) \leftarrow act(\overline{x}),
   e_{\gamma_{\phi}}(\overline{x}, \overline{w}) : \gamma(\overline{w}) \leftarrow nil.
   \]

Relation $\gamma_{\phi}$ stands for the reflexive transitive closure of $\gamma_{\phi}$ where $\phi$ holds at every configuration. More concretely, $C_1 \rightarrow_{\phi} C_n (n \geq 2)$ iff there exist $C_2, \ldots, C_{n-1}$ such that $C_1 \rightarrow_{\phi} C_2$ and $B_i \models \phi$, for $1 \leq i \leq n-1$.

**Theorem 16.** For every $B$, $A$, $\Pi$, and program $\text{Goal}(\phi_s, P, \phi_f)$, $\langle B, A, \Pi, \text{Goal}(\phi_s, P, \phi_f) \rangle \rightarrow_{\Pi} \langle B', A', \text{nil} \rangle$ if $\langle B, A, \Pi \cup \phi_s, P, \phi_f \rangle \rightarrow_{\Pi} \langle B', A', \text{nil} \rangle$.

**Proof.** Point (2) follows directly from the fact that there is no plan rule for any of the events mentioned in program $P$ or library $\Pi$. Let us know prove then point (1):

\[\text{(⇒)}\]

Once again, this theorem is a very weak notion of eliminability for the Goal-construct. However, we only need that for our purposes, namely, that …

E.4.2 Obtaining an HTN Method Library from a BDI Library with Goal-programs

We now extend the transformation $M_+^e(\Pi)$ given in Appendix E.3 to accommodate plan-bodies mentioning Goal-programs. To that end, we define the transformation $M_+^e(\Pi)$ to be exactly like $M_+^e(\Pi)$ with the following extra inductive case in the definition of $T$:

- If $P(\overline{x}) = \text{Goal}(\phi_s, P, \phi_f)$, then $T_0(P(\overline{x}), n) = \{\{\text{dummynTask}, n + 1 : \text{achieve}_{\phi_s}, n + 2 : \text{dummynTask}\}, C\}, M_+^e(\{\text{achieve}_{\phi_s} : nil \leftarrow P^* \cup \Pi^*\})$.

where $C = (n < n + 1) \land (n + 1 < n + 2) \land (n, \neg\phi_f, n + 2)^*$. and $\Pi^*$ is $\Pi$ with all $\text{Goal}(\phi_s, P, \phi_f)$ programs replaced with $P$.

Next, we generalize Theorem 15 for any kind of BDI plan library.

**Theorem 17** (Plan and HTN-Planning in Full-programs). Let $\langle N, \Lambda, \Pi, B, A, \Gamma \cup \{\text{Plan}(P)\} \rangle$ be a bounded agent, where $\Pi$ is a goal-free plan library and $P$ is a goal-free plan-body. Let $D = (Op, \emptyset \cup \{\text{dummynTask}\}, Me(\Pi))$ be the corresponding HTN problem domain and let $T_P = T_1(P; 0)$ be the network task obtained from $P$ (see as defined in Appendices E.2 and E.3). Then,

1. $\langle B, A, \text{Plan}(P) \rangle \rightarrow_{OP} \text{sol}(T_P, B, D) \neq \emptyset$.
2. $\langle B, A, \text{Plan}(P) \rangle \rightarrow_{OP} \langle B', A, \text{act}_1 \ldots \text{act}_k, \text{Plan}(P') \rangle$ with $k \geq 1$ if there exists a plan $\sigma \in \text{sol}(T_P, B, D)$, such that $\text{clean}(\sigma) = \text{act}_1 \ldots \text{act}_k$.
3. $\langle B, A, \text{act}_1 \ldots \text{act}_k, \text{Plan}(P') \rangle$ for every $\sigma \in \text{sol}(T_P, B, D)$ such that $\text{clean}(\sigma) = \text{act}_1 \ldots \text{act}_k$.

**Proof.** Follows from Theorems 15 and 16.
E.5  An Extension of HTN Semantics for Partial Plans

Here we explain that it is actually cleaner and more practical to slightly change the semantics of HTN planning to accommodate a success condition.
F. PROOF OF THEOREM 3

THEOREM 18. Let $Sim$ be a simulation relation w.r.t. relation $\overset{\text{bdi}}{\sim}$ such that $Sim(P^*, P)$. Then, program $P^*$ simulates program $P$.

PROOF. Informally, $P^*$ simulates $P$ because: (i) whenever $P$ can be terminate with internal steps, $P^*$ can also be terminated in the same way; and (ii) whenever $P$ performs a bdi-step, $P^*$ can perform the same step result, possibly by also doing some extra internal steps.

Let $E = C_1 \cdot \ldots \cdot C_n$, $n \geq 1$, be an execution of a configuration $C_1 = \langle N, \Lambda, P, B_1, A_1, \Gamma' \cup \{P\} \rangle$. We shall obtain an execution $E^*$ of $C_1^* = \langle N, \Lambda, P, B_2, A_2, \Gamma' \cup \{P\} \rangle$ such that $E$ and $E^*$ are equivalent modulo $P$ and $P^*$, and if $P$ has been successfully executed in $E$ so has $P^*$ in $E^*$.

Suppose $|E| = 1$ and hence $E = C_1$. If $P$ has not been successfully executed yet in $E$, then we take $E^* = C_2^*$ which is trivially equivalent to $E$ modulo $P$ and $P^*$ and the thesis follows. If $P$ has been executed in $E$, then $P = \overline{nil}$ and $\langle B, A, P \rangle \overset{\text{bdi}}{\sim} \langle B', A', \overline{nil} \rangle$. By point (1) in Definition 9, $\langle B, A, P^* \rangle \overset{\text{bdi}}{\sim} \langle B', A', \overline{nil} \rangle$. We can then use these internal transitions to build an execution $E'^* = C_1^* \cdot \ldots \cdot C_n^*$, $k \geq 1$, such that $P_k^* = \overline{nil}$ and $\mathcal{O}(C_1^*) = \mathcal{O}(C_2^*)$ for $i = 1 \ldots k$. Clearly, $E^* \sim E$ and $P^* \sim P$ has been executed in $E^*$.

Next suppose then $|E| = n = k + 1$, $k \geq 0$. Then, there exist an execution $E'' = C_1 \cdot \ldots \cdot C_n$ such that $|E''| = k$ and $C_1 \overset{\text{bdi}}{\sim} C_2$. We consider then the following two cases:

1. If agent transition $C_1 \overset{\text{bdi}}{\rightarrow} C_2$ is not a transition on intention $P$ itself, then it is clear that the same agent transition can be performed from $C_1^*$ to obtain $C_2^*$. That is, if $C_2 = \langle N, \Lambda, P, B_2, A_2, \Gamma' \cup \{P\} \rangle$ then $C_2^* = \langle N, \Lambda, P, B_2, A_2, \Gamma' \cup \{P^*\} \rangle$. By the induction hypothesis, there exists an execution $E''_1$ of $C_1^*$ such that $E^*_1$ and $E''_1$ are equivalent modulo $P$ and $P^*$, and if $P$ has been successfully executed in $E'$ so has $P^*$ in $E''_1$. Then, it follows that $C_1^* \cdot E^*_1$ and $C_2^* \cdot E''_1$ are equivalent modulo $P$ and $P^*$, and if $P$ has been successfully executed in $C_1^* \cdot E^*_1$ so has $P^*$ in $C_1^* \cdot E''_1$.

2. Suppose the agent transition $C_1 \overset{\text{bdi}}{\rightarrow} C_2$ is a transition on intention $P$ itself. Then, $C_2 = \langle N, \Lambda, P, B_2, A_2, \Gamma' \cup \{P\} \rangle$ such that $\langle B_1, A_1, P \rangle \overset{\text{bdi}}{\rightarrow} \langle B_2, A_2, P' \rangle$. Then, there has to exist $R'$, $R''$, $P_1'$ such that (a) $\langle B_1, A_1, P' \rangle \overset{\text{bdi}}{\rightarrow} R', R' \overset{\text{bdi}}{\rightarrow} R''$ or $R' = R''$, and $R'' \overset{\text{bdi}}{\rightarrow} \langle B_2, A_2, P'' \rangle$; and (b) $Sim(P'_1, P'')$. We use all the basic transitions from (a) to obtain an execution $E''_2 = C_1^* \cdot C_1^* \cdot C_2^*$ of $C_1$ where $j \geq 0$ and $C_2^* = \langle N, \Lambda, P, B_2, A_2, \Gamma' \cup \{P''\} \rangle$.

Because, $Sim(P'_1, P'')$ and the induction hypothesis, there exists an execution $E''_2$ of $C_1^*$ such that $E^*_2$ and $E''_2$ are equivalent modulo $P''$, and if $P$ has been successfully executed in $E'$ so has $P''$ in $E''_2$. Then, it follows that $C_1^* \cdot E''_2$ and $C_1^* \cdot C_2^* \cdot E''_2$ are equivalent modulo $P$ and $P''$, and if $P$ has been successfully executed in $C_1^* \cdot E''_2$ so has $P''$ in $C_1^* \cdot C_2^* \cdot E''_2$.

\[\square\]

Proof of Theorem 3: For every $\phi, \phi_1$ and $P$, program $\text{Plan}(\phi, P, \phi_1)$ simulates program $\text{Plan}(\text{Goal}(\phi, P, \phi_1))$.

Let $Sim(\cdot, \cdot)$ be any relation satisfying, at least, the following conditions for every $P, P', \phi, \phi_1$:
1. $Sim(\text{Goal}(\phi, P), \text{Goal}(\phi, P', \phi_1), \phi_1, \text{Plan}(\text{Goal}(\phi, P, \phi_1)))$.
2. $Sim(\text{Goal}(\phi, P), \text{Plan}(\phi, P, \phi_1)) \overset{\text{bdi}}{\rightarrow} \text{Plan}(\text{Goal}(\phi, P, \phi_1))$.
3. $Sim(\phi, \text{Plan}(\phi, P)) \overset{\text{bdi}}{\rightarrow} \text{Plan}(\text{Goal}(\phi, P, \phi_1))$.
4. $Sim(\phi, \phi_1)$.
5. $Sim(\phi, \phi_1)$.

It is hard to verify that the above relation $Sim$ is indeed a simulation relation w.r.t. relation $\overset{\text{bdi}}{\sim}$ as defined in Definition 9. By Theorem 18, the thesis follows.

\[\square\]
G. PROOF OF THEOREM 4

Lemma 20. For every agent free of \( \| \) and \( \text{Goal} \), if \( \langle B, A, \text{Plan}(P) \rangle \xrightarrow{\text{bdi}} \langle B', A', \text{Plan}(P') \rangle \), \( \langle B, A, P \rangle \xrightarrow{\text{bdi}} \langle B', A', P' \rangle \).

Proof. By assumption, we know that rule \( \text{Plan} \) was used and : (a) \( \langle B, A, P \rangle \xrightarrow{\text{plan}} \langle B', A', P' \rangle \); and (b) \( \langle B', A', P' \rangle \xrightarrow{\text{plan}} \langle B_f, A_f, \text{nil} \rangle \). Given (a) and (b), we shall prove, by induction on the number of plan-type derivation rules involved in (a), that \( \langle B, A, P \rangle \xrightarrow{\text{bdi}} \langle B', A', P' \rangle \).

Suppose that only one derivation rule is required to prove (a). Then, we have the following cases:

1. Suppose \( P = \text{act} \), \( P = \Leftrightarrow \phi \), \( P = \neg\phi \) or \( P = \neg b \). In this case, \( P' = \text{nil} \), and \( \langle B, A, P \rangle \xrightarrow{\text{bdi}} \langle B', A', \text{nil} \rangle \) follows trivially.

2. Suppose \( P = \text{nil} \). In this case, \( P' = P^n \). By hypothesis, \( \langle B, A, P \rangle \xrightarrow{\text{bdi}} \langle B', A', P^n \rangle \) follows trivially.

3. Suppose \( P = \text{null} \). In this case, \( P' = \text{null} \) and \( \langle B, A, P \rangle \xrightarrow{\text{bdi}} \langle B', A', \text{null} \rangle \) follows trivially.

4. Suppose \( P = \neg \phi \). In this case, \( P' = \neg \phi \). By directly applying rule \( \text{Plan} \), \( \langle B, A, P \rangle \xrightarrow{\text{bdi}} \langle B', A', \text{nil} \rangle \) follows directly.

Next, suppose that \( k + 1 \) derivation rules are used for (a). We then have the following cases:

1. Suppose \( P = (P_1; P_2) \). In this case, \( P \neq \text{nil} \) and \( P' = (P'_1; P'_2) \) such that \( \langle B, A, P \rangle \xrightarrow{\text{plan}} \langle B', A', P'_1 \rangle \) and \( \langle B, A, P \rangle \xrightarrow{\text{bdi}} \langle B', A', P' \rangle \) holds due to rule \( \text{Seq} \).

2. Suppose \( P = (P_1 \triangleright P_2) \). In this case, \( P \neq \text{nil} \) and \( P' = (P'_1 \triangleright P'_2) \) such that \( \langle B, A, P \rangle \xrightarrow{\text{plan}} \langle B', A', P'_1 \rangle \) and \( \langle B, A, P \rangle \xrightarrow{\text{bdi}} \langle B', A', P' \rangle \) holds due to rule \( \triangleright \).

3. Suppose \( P = \text{Plan}(P_1) \). In this case, \( P' = P'_1 \). By directly applying rule \( \text{Plan} \), we know that \( \langle B, A, P \rangle \xrightarrow{\text{bdi}} \langle B', A', P' \rangle \) and \( \langle B, A, \text{Plan}(P) \rangle \xrightarrow{\text{bdi}} \langle B', A', \text{Plan}(P) \rangle \).

This concludes the proof of the Theorem.

Lemma 21. Let \( \text{Sim}(\_ , \_ ) \) be any relation satisfying, at least, the following conditions for every program \( P \) that does not mention constructs \( \| \):
1. \( \text{Sim}(P, \text{Plan}(P)) \).
2. \( \text{Sim}(\text{nil}, \text{Plan}(\text{nil})) \).
3. \( \text{Sim}(\text{nil}, \text{nil}) \).

Then, \( \text{Sim} \) is a simulation relation w.r.t. transition \( \xrightarrow{\text{bdi}} \) for any agent that is free of concurrency.

Proof. Take a program \( P \neq \text{nil} \).

Proof of Theorem 4: Program \( P \) simulates program \( \text{Plan}(P) \) in every \( \text{CANPLAN}^+ \) agent.

Let \( \text{Sim}(\_ , \_ ) \) be any relation satisfying, at least, the following conditions for every program \( P \) that does not mention constructs \( \| \) and \( \text{Goal} \):
1. \( \text{Sim}(P, \text{Plan}(P)) \).
2. \( \text{Sim}(\text{nil}, \text{Plan}(\text{nil})) \).
3. \( \text{Sim}(\text{nil}, \text{nil}) \).

We also prove the following result.

Theorem 22. Let \( C = \langle N, \Lambda, \Pi, B, A, \Gamma \cup \{ \text{Plan}(P) \} \rangle \) be a \( \text{CANPLAN} \) agent. Let \( C' = \langle N', \Lambda', \Pi, B, A, \Gamma' \cup \{ P' \} \rangle \) where \( \Lambda', \Gamma' \) and \( P' \) are obtained from \( \Lambda, \Gamma \) and \( P \) by deleting all \( \text{Plan} \) constructs from plan-bodies. Then, configuration \( C' \) simulates configuration \( C \).
H. OTHER STUFF

LEMMA 23 (EQUAL INITIALIZATION). Let \( P_0 = \text{Plan}(\text{Goal}(\phi_s, P, \phi_f)) \), \( P_0 = \text{Goal}(\phi_s, \text{Plan}(P), \phi_f) \), and \( P^* = \text{Goal}(\phi_s, P_0, \phi_f) \), where \( P \) is a user program. Suppose that \( (B, A, P_0) \vdash C_0 \), \( (B, A, P_0) \vdash C_0 \), and \( (B, A, P^*) \vdash C^* \). Then,

1. \( C_0 = (B', A', \text{nil}) \) iff \( C_0 = (B', A', \text{nil}) \) (here, \( B \vdash \phi_f \)).
2. \( C_0 \) does not actually exist iff \( C_0 = (B, A, \text{false}) \) (here, \( B \vdash \phi_f \)).
3. \( C_0 = (B', A', P_0^*) \) where \( P_0^* = \text{Plan}(\text{Goal}(\phi_s, P_0, \phi_f)) \) iff \( C_0 = (B', A', \text{Goal}(\phi_s, \text{Plan}(P) \triangleright \text{Plan}(P), \phi_f)) \) iff \( C^* = (B', A', \text{Goal}(\phi_s, P_0 \triangleright P_0, \phi_f)) \) and for every \( C^{**} \) such that \( C^{**} \vdash C^* \), \( C^{**} = (B', A', \text{Goal}(\phi_s, P_0^* \triangleright P_0, \phi_f)) \).

4. One of the above three cases must apply for \( C_0 \).

LEMMA 24. Let \( P_0 = \text{Plan}(\text{Goal}(\phi_s, P \triangleright P_0, \phi_f)) \) and \( P^* = \text{Goal}(\phi_s, P_0 \triangleright P_0, \phi_f) \) be two programs. Then, for every \( B, A, \phi_f \),

1. \( (B, A, P_0) \vdash \text{(Plan(nil))} \) \( (B, A, \text{Plan(nil)}) \) \( (B, A, \text{nil}) \) \( (B, A, P^*) \). \( (B, A, \text{nil}) \).
2. \( (B, A, P_0) \vdash (B, A, P_0^*) \) \( (B, A, \text{Plan}(P \triangleright P_0, \phi_f)) \) \( (B, A, \text{Plan}(P \triangleright P_0, \phi_f)) \).
3. \( (B, A, P_0) \vdash (B, A, P^*) \) \( (B, A, \text{nil}) \).
4. \( (B, A, P_0) \vdash (B, A, \text{Nil}) \) \( (B, A, \text{nil}) \).

PROOF. (1) Follows directly from the fact that \( (B, A, P_0) \vdash \text{Plan(nil)} \) \( (B, A, \text{Plan(nil)}) \) \( (B, A, \text{nil}) \) \( (B, A, P^*) \). (2) Suppose that \( (B, A, P_0) \vdash (B, A, P_0^*) \) \( (B, A, \text{Plan}(P \triangleright P_0, \phi_f)) \). Then, \( (B, A, \text{Goal}(\phi_s, P \triangleright P_0, \phi_f)) \) \( (B, A, \text{Goal}(\phi_s, P \triangleright P_0, \phi_f)) \) \( (B, A, \text{Goal}(\phi_s, P \triangleright P_0, \phi_f)) \) \( (B, A, \text{nil}) \).

(3) Suppose that \( (B, A, P_0) \vdash \text{Plan(nil)} \) \( (B, A, \text{Plan(nil)}) \) \( (B, A, \text{nil}) \).

Now, if \( B \vdash \phi_f \), then \( G_T \) applies and the thesis follows trivially. Otherwise, \( B \not\vdash \phi_f \), then the only rule that may apply is \( \text{Plan(nil)} \) and the thesis follows as well.

(4) Trivial since the only derivation rules that may apply for making a Plan-transition on program \( P_0 \) are \( G_s \), \( G_T \) and \( G_S \). □

H.1 Goal and Plan Text

A central feature of CAN and \text{CANPlan} is the \text{Goal} construct for handling \textit{declarative} goals. It is important then to understand the interaction of goal-programs with the new planning construct \text{Plan}. That is, we want to explore the differences and similarities between “having the goal to plan” (i.e., \( \text{Goal}(\phi_s, \text{Plan}(P), \phi_f) \)) and “planning for a declarative goal” (i.e., \( \text{Plan}(\text{Goal}(\phi_s, P, \phi_f)) \)).

To begin with, it is easy to prove that both programs are “initialised” in equivalent ways: the first goal-plan-program is initialised to program \( \text{Goal}(\phi_s, \text{Plan}(P) \triangleright \text{Plan}(P), \phi_f) \) iff the second plan-goal-program is initialised to program \( \text{Plan}(\text{Goal}(\phi_s, P \triangleright P, \phi_f)) \).

Now, a goal-plan program of the form \( \text{Goal}(\phi_s, \text{Plan}(P), \phi_f) \) has the same meaning as the one provided in CAN: execute \( \text{Plan}(P) \) until the goal \( \phi_s \) is achieved or the failure condition \( \phi_f \) applies. Note that nothing precludes solving \( P \) completely without achieving \( \phi_s \). Still, due to the persistence property of \text{Goal}, the whole goal will be retrieved if \( \text{Plan}(P) \) fails or does not achieve \( \phi_s \). In addition, \text{Plan} \( P \) need not be executed completely, that is, a partial execution of \( P \) that would achieve \( \phi_s \) would not work.

Consider next its plan-goal counterpart \( \text{Plan}(\text{Goal}(\phi_s, P, \phi_f)) \). Such a program is closer to a \textit{classical planning problem}: plan for achieving the declarative goal \( \phi_s \) (by using the given program \( P \)). If \( P \) cannot be solved or it is not able to achieve \( \phi_s \), then the whole plan-goal will fail. Thus, we are merely interested in solving program \( P \), but also doing it in a way that will in fact achieve the desired goal \( \phi_s \). In addition, program \( P \) need not be executed completely but only to the point where \( \phi_s \) is realized. This type of behaviour on (declarative) goals does not exist in CAN. It is thus sensible, for convenience, to define a new language construct \( \text{Plan}(\phi, P) \) \( \text{Plan}(\phi, P, \phi_f) \): plan for achieving \( \phi \) using \( P \). Notice that \( \text{Plan}(\phi, P) \) is close but more expressive than program \( P(\phi) \)—the latter requires \( P \) to be completely executed.

We summarize the differences between goal-plan and plan-goal programs as follows:

<table>
<thead>
<tr>
<th>Goal(\phi_s, Plan(P), \phi_f)</th>
<th>Plan(\phi_s, P, \phi_f)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BDI</strong> execution driven ✓</td>
<td><strong>classical planning driven ✓</strong></td>
</tr>
<tr>
<td>P may achieve ( \phi_s ) ✓</td>
<td>P must achieve ( \phi_f ) ✓</td>
</tr>
<tr>
<td>P may be re-tried ✓</td>
<td>P is never re-tired</td>
</tr>
<tr>
<td>P must execute fully ✓</td>
<td>P may execute partially ✓</td>
</tr>
<tr>
<td>Fails if ( \phi_f ) holds</td>
<td>Fails if ( \phi_f ) holds</td>
</tr>
</tbody>
</table>

What we would like is a program that combines the advantages of both versions. That is, we may want to deliberate and act towards a clear goal \( \phi_s \) using a program \( P \) and be committed to the goal until it is either achieved or impossible. Interestingly, program \( \text{Plan}(\text{Goal}(\phi_s, P, \phi_f)) \) has such properties and it will (a) always execute towards goal \( \phi_s \); (b) terminate when the goal is achieved or is impossible; and (c) re-try and insist on program \( P \) if necessary.

THEOREM 25. Let \( C_0 = (\Lambda', \Pi, B, A, \Gamma_0 \cup \{P_0^X\}) \) be an agent configuration with \( X \in \{a, b\} \), where \( P \) is a user program

\[ P_0^X = \text{Plan}(\text{Goal}(\phi_s, P, \phi_f)), \]

\[ P_0^X = \text{Goal}(\phi_s, \text{Plan}(P), \phi_f), \]

Let \( E = C_0 \vdash \cdots \vdash C_n \) be an execution of \( C_0 \) such that \( P_0^X \) is currently executing in \( E \) (i.e., \( P_0^X \neq \emptyset \)). Next, let us consider \( C_0^X = (\Lambda', \Pi, B, A, \Gamma_0 \cup \{P_0^X\}) \), where \( P_0^X = \text{Goal}(\phi_s, P_0^X, \phi_f) \).

Then, there exists an execution \( E^* = C_0^X \vdash \cdots \vdash C_m^X \) of \( C_0^X \) such that (a) \( |E| \) and \( |E^*| \) are equivalent modal intentions \( P_0^X \) and \( P_0^X \); and (b) intention \( P_0^X \) has been successfully executed in \( E \) iff intention \( P_0^X \) has been successfully executed in \( E^* \).
Let $P_a = \text{Plan}(\text{Goal}(\phi_s, P, \phi_f))$ and let $P_b = \text{Goal}(\phi_s, \text{Plan}(P), \phi_f)$. Suppose $E = \langle C_0, \ldots, C_n \rangle$ is an execution of $C_0 = \langle N, A, B, A, \Gamma \cup \{ P_a \} \rangle$ such that $P_a$ is currently executing in $E$. Then, there exists an execution $E' = \langle C_0', \ldots, C_n' \rangle$ of $C_0' = \langle N, A, B, A, \Gamma \cup \{ P_b \} \rangle$ such that $E'$ and $E$ are equivalent and

1. $\Gamma_n = \Gamma^* \cup \{ P_n \}$ if $\Gamma'_n = \Gamma^* \cup \{ P'_n \}$;
2. if $P_n = \text{Plan}(\text{Goal}(\phi_s, P' \triangleright P, \phi_f))$, then $P'_n = \text{Goal}(\phi_s, \text{Plan}(P') \triangleright \text{Plan}(P), \phi_f)$;
3. if $P_n = \text{Plan}(\text{nil})$, then $P'_n = \text{nil}$;
4. if $(B_n, A_n, P_n)$ $\text{bdi} \rightharpoonup$ and $P_n \neq \text{nil}$, then $P'_n = \text{Plan}(\text{Goal}(\phi_s, P' \triangleright P, \phi_f))$ and $(B_n, A_n, \text{Plan}(P'))$ $\text{bdi} \rightharpoonup$.

The proof relies on a few lemmas; the most important of them states that for any $B, A, P, \phi_s, \phi_f$, and $P \neq \text{nil}$: if $(B, A, \text{Plan}(\text{Goal}(\phi_s, P \triangleright P_o, \phi_f)))$ can make a bdi-type transition to $(B', A', \text{Plan}(\text{Goal}(\phi_s, P' \triangleright P_o, \phi_f)))$, then it is the case that $(B, A, \text{Goal}(\phi_s, \text{Plan}(P') \triangleright \text{Plan}(P_o), \phi_f))$ can make a bdi-type transition to $(B', A', \text{Goal}(\phi_s, \text{Plan}(P') \triangleright \text{Plan}(P_o), \phi_f))$.

The reader may wonder if it is possible to combine the features of both type of programs. That is, one may want to deliberate on, possibly partially, doing $P$ in a way that will indeed achieve $\phi_s$ while allowing for the possibility of re-trying $P$ if an external event interferes with the solution found at planning time. Interestingly, the program we are looking for is $\text{Goal}(\phi_s, \text{Plan}(\text{Goal}(\phi_s, P, \phi_f)), \phi_f)$. To show this, we first need to abstract away from some irrelevant internal steps in an agent execution. If $E = C_0 \cdots C_n, n \geq 0$, is an agent execution, then $C_0 \cdots C_n$ is the sequence obtained from $E$ by deleting all elements $C_j$ of the sequence such that the $C_j \Rightarrow C_{j+1}$ step used in $E$ is due to the agent derivation rule $A_{step}$ and either of the basic derivation rules $G_I$ or $\text{Plan}_t$. 