ABSTRACT

Belief-Desire-Intention (BDI) agent systems and Hierarchical Task Network (HTN) planning systems are two popular approaches to acting and planning, both of which are based on hierarchical and context-based expansion of subgoals. Over the past three decades, various authors have recognised the similarities between the two approaches, and developed methods for making the domain knowledge embedded in one system accessible to the other, and for augmenting BDI agents with the ability to perform HTN-style “lookahead” planning. This paper makes a novel contribution to this strand of work by developing a formal account of “plugging” in available HTN hierarchies (e.g. from the International Planning Competition) into a BDI agent’s goal-plan hierarchy. When combined with lookahead-based execution, the agent is then guaranteed to behave in accordance with the “operational guidelines” embedded in the HTN hierarchies. We also explore how HTNs could be used to obtain BDI hierarchies that can be executed without performing any lookahead. In particular, we first characterise a useful class of BDI agent hierarchies that any such translation should produce, and then characterise the restrictions that need to be imposed on HTNs in order to encode them as useful BDI hierarchies.

1. INTRODUCTION

Belief-Desire-Intention (BDI) agent systems [18] and Hierarchical Task Network (HTN) planning systems [8] are well-understood and successful approaches to acting and planning. Both of these approaches are based on hierarchical and context-based expansion of subgoals: while BDI agents interleave this process with acting in order to be responsive to environmental changes, HTN planners perform complete “lookahead” over subgoal expansions in order to guarantee that they are achievable. Over the past three decades, various authors have recognised, to different extents, the similarities between the two approaches (e.g. [3, 15, 23, 9, 6, 20, 19]). This has led to methods for making the (operator-supplied) domain knowledge that is available in one system accessible to the other, and for augmenting BDI agents with the ability to perform HTN-style (complete) lookahead over their goal-plan hierarchies.

This paper makes a novel contribution to this strand of work by developing a formal account of “plugging” available HTN hierarchies, e.g. from the International Planning Competition [5] or real-world HTN planning applications [13], into a BDI agent’s goal-plan hierarchy. These may either represent new strategies/procedures for achieving an agent’s existing goals, or represent a collection of procedures for handling new goals altogether. When the plugged hierarchies are combined with the lookahead-based acting semantics of [19], the agent is guaranteed to behave in accordance with the “operational guidelines” that are in the original (HTN) hierarchies. In general, lookahead-based acting allows an agent to deliberate over the outcomes of making one choice (e.g. regarding how to decompose a subgoal) over another. Such deliberation is sometimes necessary for avoiding undesired situations and guaranteeing goal achievability, such as when irreversible actions are present that may lead to “dead end” states—from where there is no successful outcome; action execution takes significantly longer than lookahead deliberation; or actions may consume important resources [19, 20]. We also explore how HTNs could be used to obtain BDI hierarchies that can be executed without needing to perform any lookahead, and we study the resulting tradeoffs.

In past work, the first systematic study of the similarities and differences between HTN and BDI systems is presented in [6]. In particular, the authors map from an existing HTN Blocks World domain into an equivalent BDI agent domain, and provide preliminary results which suggest that the HTN hierarchies can then be executed more effectively by an agent when the environment is dynamic. Unlike our work, [6] uses a specific implementation of each system (JSHOP [14] and JACK [2], respectively), rather than their formal syntax and semantics. The first formal mapping from one system to the other is proposed in [19, 20] with the CANPlan framework. CANPlan is an extension of the CAN [24] BDI agent programming language to include HTN planning is a built-in feature, allowing an agent to perform lookahead deliberation from user-defined points in the agent’s goal-plan hierarchy. To this end, the authors show how CANPlan (recipe) libraries can be converted into equivalent HTN domains, though not the other way around as we do in this work. It turns out that converting HTN domains into CANPlan libraries requires a different approach to the one in [19, 20], particularly because HTNs employ certain constraints which have no direct counterparts in CANPlan (or CAN).

The contributions of this paper are twofold. First, we propose a novel translation from HTN domains into CANPlan agent libraries. We show that by performing lookahead-based execution on the resulting recipes, the agent will conform to the “operational guidelines” embedded in the HTN domain. To this end, we use the notion of a “declarative goal”, which is central in CAN, and we briefly describe a necessary alternative to the operational semantics of declarative goals in the context of planning. Second, we explore the tradeoffs in translating HTN domains into the traditional (non-planning) AgentSpeak-like, CAN libraries. In particular, we first characterise a useful class of CAN agent library that any such
translation should produce, and we characterise the restrictions that need to be imposed on HTN domains in order to encode them as useful CAN libraries.

2. BDI AND HTN SYSTEMS

This paper uses the CAN and CANPlan syntax and semantics from [20], and the most general HTN planning syntax and semantics from [8]. We summarise these formalisms below, and touch upon their similarities.

2.1 BDI Agent Programming Languages

A CAN or CANPlan BDI agent is created by specifying a belief base, $B$, i.e., a set of ground atoms; a plan-library, $P$, i.e., a set of plan-rules; and an action-library, $\Delta$. A plan-rule is of the form $e(t) \vdash \psi \leftarrow P$, where $e(t)$ is an event-goal, $t$ is a vector of terms; and $\psi$, a formula, is the context condition. The plan-body $P$ is built from: actions $\text{act}(t)$; belief addition $++, b$ and removal $--, b$ programs which are used to respectively add and remove atom $b$ from $B$; test programs, $? \phi$, where $\phi$ is a formula, which are used to test whether $\phi$ holds in $B$; event-goal programs $?e$, where $e$ is an event-goal; and declarative goal programs $\text{Goal}(\phi, P, ? \phi)$, specifying that formula $\phi$ (the declarative goal) should be achieved using program $P$, failing if $\phi$ becomes true. As a plan-body “evolves”, the following “internal” constructs may also be used: programs $nil$, $P_1 \triangleright P_2$, and $\{\psi_1 : P_1, \ldots, \psi_n : P_n\}$. Intuitively, $nil$ indicates that there is nothing left to execute; program $\{\psi_1 : P_1, \ldots, \psi_n : P_n\}$ is the set of plan-rules that are relevant for an event-goal; and $P_1 \triangleright P_2$ captures failure recovery: $P_1$ should be tried first, failing which $P_2$ should be tried. Formally, a CAN plan-body is described by the grammar

$$P ::= \text{nil} \mid \text{act} \mid ? \phi \mid ++b \mid --b \mid t_1 : P_1 \triangleright P_2 \mid \{\psi_1 : P_1, \ldots, \psi_n : P_n\} | \text{Goal}(\phi, P, ? \phi).$$

The above grammar also describes a CANPlan plan-body when construct $\text{Plan}(P)$ is included, specifying that program $P$ should be executed only if it has a successful HTN decomposition.

The transition relation on a CAN configuration is defined as a set of derivation rules in the style of [16]. A derivation rule has an antecedent and a conclusion: the latter is a (single) transition, and the former can be empty or have transitions and auxiliary conditions. A transition $C \rightarrow C'$ denotes that configuration $C$ yields configuration $C'$ in a single execution step. A configuration is a tuple $(\langle B, A, P \rangle, \Delta)$, where $A$ is the sequence of actions executed so far, and the other elements are as above; when comparing configurations, we ignore the static elements $P$ and $\Delta$, and we sometimes omit them for brevity. CANPlan uses labelled transitions denoted $C \xrightarrow{\text{lab}} C'$, where $\text{lab} \in \{\text{bd}, \text{plan}\}$, and when there is no label on a transition both types apply. Intuitively, bdi-type transitions are used for standard BDI reasoning and acting, and plan-type transitions for “internal” steps within a planning context. These transition types allow precluding certain rules, e.g. those capturing BDI failure recovery, from being applied in a planning context.

For example, consider the two CANPlan derivation rules below. In rule $D_1$, configuration $(\langle B, A, \{\Delta\} \rangle)$—with only an update to the last component—in one transition of type bdi or plan. Construct $\text{mgu}$ stands for “most general unifier” [111], and $\{\Delta\}$ is the possibly empty set of relevant plan-rules for $e$, i.e., the ones whose associated event-goal unifies with $e$. Rule $D_2$ states that any (single) bdi-type execution step on $\text{Plan}(P)$ necessitates one or more (internal) steps that yield a successful HTN decomposition of $P$.

$$\Delta = \{\psi, \theta : P_1, \theta' : \psi' \leftarrow P_1 \in \Pi \land \theta = \text{mgu}(\psi, \psi')\}$$

$$\langle B, \Delta \rangle \xrightarrow{\text{plan}} (\langle B, \{\Delta\} \rangle)$$

$$\langle B, \Delta \rangle \xrightarrow{\text{bdi}} (\langle B, \{\Delta\} \rangle)$$

Finally, the action-library $\Delta$ is a set of action-rules which, like STRIPS operators, are of the form $\text{act}(v) : \psi \leftarrow \Phi^+ \land \Phi^-$, where $\text{act}(v)$ is an action; $v$ is a vector of distinct variables; $\psi$ is as above; and $\Phi^+$ and $\Phi^-$ are the add-list and delete-list, respectively. All variables appearing in the rule must also occur in $v$, and any given action has at most one associated action-rule in $\Delta$.

2.2 HTN Planning

Like BDI agent reasoning, HTN planning is the process of decomposing, from an initial state, the compound tasks in an initial task network, until only primitive tasks remain. However, while HTN systems are concerned with hypothetical offline reasoning about actions and their potential interactions within a pursued plan for solving a task, BDI systems focus on the effective online execution of plans, in complex and dynamic environments.

Formally, an HTN planning problem is a tuple $(d, I, \Delta)$, where $d$ is a task network and $I$ is an initial state (set of ground atoms). Element $D = (Op, Me)$ is an HTN (planning) domain, where $Op$ is a set of STRIPS-like operators and $Me$ is a set of methods. Elements $d, I, Op$ and $Me$ are comparable to elements $P, B, \Delta$, and $\Pi$ above (respectively).

A task network $d$ is a tuple $[\text{S}, \phi]$, where $S$ is a non-empty set of elements of the form $(n : \tau(t))$: element $n$ is a task label, which is unique in the planning problem; and $\tau(t)$ is either a compound task or primitive task. Element $\phi$, the task network formula, is a Boolean formula built from negation, conjunction, disjunction, and the following constraints: variable binding constraints of the form $(t = t')$, requiring $t$ and $t'$ (function-free terms) to be equal; ordering constraints of the form $(n < n')$ (sometimes with brackets omitted), requiring task label $n$ to precede task label $n'$; before (resp. after) state constraints of the form $(l, n)$ (resp. $(n, l)$), requiring literal $l$ to hold (in the state) immediately before (resp. after) label $n$; and between state constraints of the form $(n, l, n')$, requiring literal $l$ to hold in all states between labels $n$ and $n'$ (the constraint holds vacuously if no such states exist).

Like a CAN action, an HTN primitive task has at most one associated operator in $Op$, and like a CAN event-goal (or event-goal program), a compound task can have more than one associated method in $Me$. A method is a tuple $[\tau, d]$, where $\tau$ is a compound task and $d$ a task network. An example of a set of methods—which we use as the running example in this paper—is shown below; all tasks are primitive except for $\tau_1, \tau_3$ and $\tau_4$.

$${\tt m}_1 = \{\tau_1, \langle(2 : \tau_2(Y), (3 : \tau_3)), 2 < 3 \land (q(Y), 2)\}\}$$

$${\tt m}_2 = \{\tau_3, \langle(4 : \tau_4, (5 : \tau_5)), 4 < 5 \land (4, p), 5)\}\})$$

$${\tt m}_3 = \{\tau_4, \langle(7 : \tau_7, (8 : \tau_8)), 7 < 8 \land (q(2), 7)\}\})$$

$${\tt m}_4 = \{\tau_4, \langle(9 : \tau_9), \text{true}\}\})$$

Given an HTN planning problem $(d = [S_d, \phi_d], I, D)$, with $D = (Op, Me)$, HTN planning involves selecting a reduction method $m = [\tau_m, d_m] \in Me$ and applying it to a compound task $(n : \tau) \in S_d$ (provided $\tau$ and $\tau_m$ unify) to yield a new, and typically “more primitive” task network $d'$. Formally, this is denoted by $d' = \text{reduce}(d, n, m)$, and the set of all reductions of $d$ is denoted by $\text{red}(d, D)$. Applying a reduction to $(n : \tau)$ above involves updating the set $S_d$ by replacing $(n : \tau)$ with the tasks in $d_m$, and
The new rule (D3) and the adapted one (D4) are shown below.

\[
\begin{align*}
\langle B, A, \text{Goal}(\phi_s, l_e, \phi_f) \rangle & \xrightarrow{\text{plan}} \langle B, A, (l_e; ?\phi_s) \rangle \\
B \not= (\phi_s \lor \phi_f) & \xrightarrow{\text{bdi}} \langle B', A', P \rangle \\
\langle B, A, \text{Goal}(\phi_s, l_e, \phi_f) \rangle & \xrightarrow{\text{bdi}} \langle B', A', \text{Goal}(\phi_s, P \triangleright P', \phi_f) \rangle
\end{align*}
\]

In rule D4, program P is the original set of relevant plan-rules for event-goal program le, which is “copied” during goal adoption and persistently re-instantiated by another derivation rule if P eventually fails (i.e., it becomes “blocked”).

The authors point out, however, that another possible semantics for declarative goals in the context of planning is where the failure condition is used as declarative “control information”. Since such a semantics is crucial for our translation from HTNs to BDI agent recipes, we shall briefly describe the main changes needed. First, we remove rule D3, and remove the labels on D1’s transitions, so that all derivation rules defined for declarative goals may be used in a planning context. To be consistent with the rest of the semantics, however, we preclude the “goal restart” rule—which recovers from a “blocked” program P by retrying the original program P—from being used in a planning context, by replacing its transitions with bdi-type ones. The resulting rule is shown below.

\[
\langle B, A, P \rangle \xrightarrow{\text{bdi}} \langle B', A', \text{Goal}(\phi_s, P' \triangleright P', \phi_f) \rangle
\]

Finally, the assumptions that we make in this paper are as follows. First, intuitively, we assume that any given HTN domain “works” in at least one situation, or in other words, that it is internally consistent. Formally, given an HTN domain \(D = (\Omega, Me)\), for any method \([\tau, d] \in Me\), there exists an initial state \(T\) such that \(sol(d, T, D) \neq \emptyset\). Second, we assume without loss of generality that any variable occurring in a task network \(d = [S, \phi]\) is mentioned at the “start” of \(d\). Formally, there exists a task \((n : \tau') \in S\) such that (i) \(\phi \supset (n < n')\) for all \((n' : \tau') \in S\) with \(n \neq n'\); and (ii) any variable occurring in \(d\) also occurs in some before constraint \((l, n)\) with \(\phi \supset (l, n)\).1

Basically, this assumption amounts to ensuring that any declarative goal will be grounded at the time it is adopted by the agent, as required in [20].

### 4. HTNs TO PLANNING AGENTS

This section develops mechanisms for essentially “plugging in” a given HTN domain into a CANPlan agent’s plan-library. By performing lookahead-based execution on the resulting rules, the agent is then guaranteed to behave in accordance with the HTN constraints that are specified in the HTN domain.

We start by defining a class of HTN domain called a conjunctive HTN domain. These have simpler constraint formulas which are easier to convert into BDI entities, but are no less expressive than standard HTN constraint formulas.

**Definition 2 (Conjunctive HTNs).** An HTN task network \([S, \phi]\) is conjunctive if its formula \(\phi\) is a conjunction of possibly negated HTN constraints. An HTN domain \((\Omega, Me)\) is conjunctive if the task network \(d\) in every method \([\tau, d] \in Me\) is conjunctive.

In what follows, we assume that all HTN task networks and domains are conjunctive. The following theorem states that any HTN

1When checking material implication, we treat HTN constraints as propositions.
domain has at least one conjunctive counterpart, and that by using a conjunctive domain we do not lose generality. In the following theorem and later in the paper, we sometimes assume for simplicity (and WLOG) that the task network to be solved (i.e., the first element in the planning problem) has only one task, as it can always be replaced with a more “complex” task network via a reduction.

**Theorem 1.** Let \( D = \langle Op, Me \rangle \) be an HTN domain. There exists a conjunctive HTN domain \( D' = \langle Op, Me' \rangle \) such that for any labelled task \( (n : \tau) \) and initial state \( T, sol(d, I, \tau, D) = sol(d, I, \tau, D') \), where \( d = \{(n : \tau), true\} \). ■

Given an HTN domain \( D' \) as above, we obtain a conjunctive domain \( D = \langle Op, Me \rangle \) as follows. For each method \( \langle \tau, [S, \phi] \rangle \in Me \), we add the method \( \langle \tau, [S, \phi'] \rangle \) to \( Me' \) (which is initially empty) for each disjunct \( \phi' \) appearing in \( \phi_{\text{refl}} \), where \( \phi_{\text{refl}} \) is in disjunctive normal form.

Next, we define some auxiliary notions that are needed for our translation. First, we define the notion of a set of terminating tasks of a given compound task. Intuitively, this set represents one possible sequence of decompositions of the compound task into a primitive task network.

**Definition 3 (Terminating Tasks).** Let \((n : \tau)\) be a compound labelled task and \( D = \langle Op, Me \rangle \) an HTN domain. If there exists a method \( \langle \tau', [S, \phi] \rangle \in Me \) such that \( \tau' \) and \( \tau \) are the same type,\(^2\) then let \( S_1 \) (resp. \( S_2 \)) be the set of labelled primitive (resp. compound) tasks in \( S \); otherwise, let \( S_1 = S_2 = \emptyset \). A set of terminating tasks of \((n : \tau)\) relative to \( D \), denoted \( \text{FIN}\{ (n : \tau), D \} \), is defined inductively as

\[
S_1 \cup \bigcup_{i'(n : \tau')} \text{FIN}\{ (n : \tau'), D \}.
\]

The set of all terminating tasks of \((n : \tau)\) relative to \( D \), denoted by \( \text{FIN}^*( (n : \tau), D ) \), is the smallest set such that for any set \( \text{FIN}\{ (n : \tau), D \} \), the set is in \( \text{FIN}^*( (n : \tau), D ) \).

Observe that \( \text{FIN}^* ( (n : \tau), D ) \) is a set of sets. Given a primitive labelled task \((n : \tau)\), we define \( \text{FIN}\{ (n : \tau), D \} = \{(n : \tau)\} \), and \( \text{FIN}^* ( (n : \tau), D ) = \{( (n : \tau) \} \} \).

Given a task, we use its set of all terminating tasks to determine whether its corresponding BDI entity (action or event-goal) has completed execution. In particular, we query the agent’s belief base to check whether one of the sets of terminating tasks (actions) corresponding to the event-goal has been executed. To this end, the agent’s belief base keeps track of all the actions executed so far, as well as the sequence of decompositions—or “path”—that led to an action, via the distinguished function symbol \( f \). Formally, a path function (or path) is defined inductively as the 2-ary (FOL) function symbol \( f(t_1, t_2) \), where \( t_2 \) is a variable or task label, and \( t_1 \) is either the distinguished constant \( \text{top} \) or a path function. For example, if task label \( l_3 \) corresponds to an action, then \( f(f(f(\text{top}, l_3), n_2), n_3) \) represents two decompositions of the top-level event-goal (compound task) corresponding to label \( l_1 \). We sometimes omit the arguments of a path function when they are not needed.

We define the prefix and end of a given path with the following three axioms (i.e., Horn clauses as in [1]), and assume that they are taken into account when the agent queries its belief base \( B \).\(^3\)

---

\(^2\)i.e., the two tasks have the same symbol and arity

\(^3\)CAN simply assumes that an operation exists for checking whether a condition \( \phi \) follows from a belief base \( B \) [20]. Thus, we assume that an operation \( B \models \phi \phi \) is provided, where \( A = \{A_1, A_2, A_3\} \) is our set of axioms.
with each \( P_i \) as defined above. Informally, the declarative goal can be accomplished if \( p(X) \) can eventually be made to hold, i.e., all parallel steps occurring in \( P \) can be successfully interleaved and completed, but without ever making failure condition \( \phi_f \) hold. Intuitively, combining this declarative goal with the lookahead construct (\texttt{Plan}) allows “monitoring” the program \( P \) being pursued and “backtracking” when a step makes \( \phi_f \) true, which amounts to violating a constraint in the HTN method’s constraint formula.

For example, the translation in this section produces the following plan-rules, from the methods introduced in Section 2.2.4:

\[
\begin{align*}
    r_1 &= \langle s_1 \leftarrow q(Y) \rangle \leftarrow \text{Goal}(\phi_{s_1}, P_1, \phi_{f_1}) \\
    r_2 &= \langle s_3 \leftarrow \text{true} \rangle \leftarrow \text{Goal}(\phi_{s_2}, P_2, \phi_{f_2}) \\
    r_3 &= \langle s_4 \leftarrow q(2) \rangle \leftarrow \text{Goal}(\phi_{s_3}, P_3, \phi_{f_3}) \\
    r_4 &= \langle s_4 \leftarrow \text{true} \rangle \leftarrow \text{Goal}(\phi_{s_4}, (a_5 + \phi_{s_4}), \text{false})
\end{align*}
\]

where

\[
\begin{align*}
P_1 &= \langle a_2(Y) \parallel a_3 \rangle + \phi_{s_1} \\
P_2 &= \langle a_4 \parallel a_5 \parallel \phi_{s_2} \\
P_3 &= \langle a_7 \parallel \phi_{s_3} \rangle + \phi_{s_3}
\end{align*}
\]

4.2 HTN Constraints as Failure Conditions

We take the failure condition \( \phi_f \) as the negation of the formula \( \text{TRANS}(c_1) \land \ldots \land \text{TRANS}(c_h) \), where each \( c_i \) is a conjunct in the HTN constraint formula \( \phi_{\text{htn}} \) above, and \( \text{TRANS}(c_i) \) is defined below. In what follows, we ignore the HTN domain \( D \); e.g., we write \( \Phi^1(f) \) instead of \( \Phi^1(f, D) \). Let us now consider the possible values for a conjunct \( c \) in \( \phi_{\text{htn}} \).

- **Conjunct \( c = (n_1 < n_2) \), i.e., \( c \) is an ordering constraint.** Then, \( \text{TRANS}(c) \) is the following formula, which requires all actions of \( n_2 \) to have completed at some point before the first action of \( n_2 \) starts:

  \[
  \Phi^1(f_{n_1}) \models \Phi^1(f_{n_2})
  \]

  where \( f \) denotes \( f(X, i) \). Strictly speaking, the left hand side of the condition checks whether the first action of \( n_2 \) has completed (as opposed to started), but this is sufficient because (i) an action completes in one (CAN) step; (ii) the condition will be checked at every step; and (iii) actions cannot overlap (i.e., they are interleaved).

- **Conjunct \( c = (n_1, l, n_2) \), i.e., \( c \) is a between state constraint.** Then, \( \text{TRANS}(c) \) is the following formula, which requires literal \( l \) to hold from just after the last action of \( n_1 \) until just before the first action of \( n_2 \):

  \[
  \left( \Phi^1(f_{n_1}) \models l \right) \lor \Phi^1(f_{n_2})
  \]

- **Conjunct \( c = (l, n) \), i.e., \( c \) is a before state constraint.** Then, \( \text{TRANS}(c) \) is the following formula, which requires literal \( l \) to hold until just before the first action of \( n \):

  \[
  \left( \Phi^1(f_{n}) \models l \right) \lor \Phi^1(f_{n})
  \]

Informally, task label \( n' \) represents the primitive task (action) that precedes \( n \) in the above method’s task network. We leave out the formal definition for brevity. The translation of an after state constraint, where \( c = (n, \bar{l}) \), is analogous to the translation above.

We translate negated HTN constraints as follows. If \( c \) is \( \neg(l, n) \) or \( \neg(n, l) \), we define \( \text{TRANS}(c) \) as respectively \( \text{TRANS}((n, \neg l)) \) or \( \text{TRANS}((\neg l, n)) \). If \( c = \neg((n_1 < n_2)) \), then \( \text{TRANS}(c) \) is the converse of the formula corresponding to the case where \( c = (n_1 < n_2) \), as we must now check that at least one of the actions of \( n_2 \) complete before all of the actions of \( n_1 \) complete. For example, the failure conditions \( \phi_{f_1}, \phi_{f_2}, \phi_{f_3} \) in the declarative goals of our running example are the negations of the following formulas, respectively:

\[
\begin{align*}
\Phi^1(f_1) &\models \Phi^1(f_2), \quad (\Phi^1(f_4) \models p) \lor \Phi^1(f_5), \quad \Phi^1(f_6) \models \Phi^1(f_7)
\end{align*}
\]

Using the described translation from HTN methods to BDI plan-rules, we can show that an HTN planner will find a solution for a labelled task \( n: \tau(t) \) if and only if the corresponding CANPlan agent finds the same solution by performing lookahead on the corresponding event-goal program \( \tau(t \cdot f(top, n)) \), which we denote as \( l_{\text{can}} \). In the theorem below, subscripts denote the HTN entries that were used to obtain the corresponding CANPlan entities; e.g., \( \Pi_{\text{can}} \) is the CANPlan plan-library obtained from the HTN method-library \( Me \), via the translation above.

**Theorem 2.** For any HTN domain \( D = \langle Op, Me \rangle \), initial state \( I \), task network \( d = \{ (n: \tau), \text{true} \} \), and action sequences \( \sigma, A \):

\[
\sigma \in \text{sol}(d, I, D) \iff \sigma \in \text{sol}(\Pi_{\text{can}}, \Lambda_{\text{can}}, B_{\text{can}}, A, \text{Plan}(l_{\text{can}})) \text{ bdi} \iff \langle I_{\text{can}}, \Lambda_{\text{can}}, B_{\text{can}}, A \cdot \sigma, \text{nil} \rangle.
\]

Thus, any sequence of actions \( \sigma \) executed by the agent from belief base \( B_{\text{can}} \), in order to achieve event-goal \( l_{\text{can}} \) via the libraries \( \Pi_{\text{can}} \) and \( \Lambda_{\text{can}} \) (translated from \( D \)), will be an HTN solution for the corresponding HTN planning problem, and vice versa. The proof of this theorem is involved, and is based on induction on both the structure of HTN methods and the length of HTN and CANPlan decomposition sequences. Informally, the main step is to show that a “successful” sequence of HTN reductions of a task network corresponds to a sequence of CANPlan configurations \( C_1 \cdot \ldots \cdot C_h \), such that for each \( C_i \), no adopted (and possibly “nested”) declarative goal appearing in \( P_i \) has a failure condition that holds in \( B_i \), (where \( P_i \) and \( B_i \) is the plan-body and belief base in \( C_i \), respectively.) We shall illustrate this main step in the example below.

4.3 An Example

As an example of how the failure conditions of declarative goals are monitored, let us once again consider the HTN methods and resulting BDI plan-rules from our running example. Let us also suppose that the primitive task \( \tau_2(Y) \) removes atom \( q(Y) \) and adds atom \( p \), and that tasks \( \tau_5 \) and \( \tau_9 \) remove atom \( p \).

Consider a possible lookahead-based execution of event-goal program \( \text{le}_1 \), i.e., a particular execution trace of the CANPlan program \( \text{Plan}(\text{le}_1, t, f(top, 1)) \), from a belief base \( B_0 \models q(1) \land q(2) \). First, plan-rule \( \tau_1 \) is selected to achieve \( \text{le}_1 \); substitution \( \{ Y/1 \} \) is applied to its context condition, and the associated declarative goal is adopted via the rule described in Section 3. The latter is possible because the current belief base does not entail the success condition \( \phi_{s_1} \) nor the failure condition \( \phi_{f_1} \). These must not be entailed by the belief base as the goal progresses either, and the same holds for any other (possibly nested) declarative goals that are adopted.

Next, action \( a_{\tau_2}(1) \) is chosen (arbitrarily) from the parallel steps in \( P_1 \); resulting in atom \( q(1) \) being removed from \( B_0 \), and atom \( p \)

\(^3\)The case where \( c = (n_1, l, n_2) \) — i.e., \( \neg l \) must hold somewhere between \( n_1 \) and \( n_2 \) — requires a more involved translation which we have left out for brevity.

\(^4\)Conversely to the above result, [20] shows that CANPlan libraries can be translated into equivalent HTN entities. However, the proof of our theorem follows a similar approach to theirs.
and the “path predicate” \( p(f(f(t_0, 1), 2)) \) being added to it, yielding the updated belief base \( B \). Following this, plan-rule \( r_2 \) is selected to achieve event-goal program \( \lambda_3 \), and plan-rule \( r_3 \) is selected to achieve \( \lambda_4 \), resulting in their associated declarative goals also being adopted.

In our scenario, the next action that is chosen is \( a_7 \), adding atom \( p(f(f(f(t_0, 1), 3), 4), 7) \) to \( B \). At this point, \( B \) still does not entail \( \phi_3 \); while \( \lambda_3 \) did indeed just execute its first action, this was done only after \( a_7 \). Similarly, action \( a_8 \) is chosen next, adding its associated path predicate to \( B \). At this point, \( B \) still does not entail \( \phi_3 \); action \( a_8 \) was indeed just executed, but only after \( a_7 \). Moreover, \( B \) still does not entail \( \phi_3 \) either: all actions associated with \( \lambda_4 \) were just executed, but proposition \( p \) also holds in \( B \) (due to \( a_2 \)). The belief addition \( \phi_3 \) is then executed with \( \phi_3 = p(f(f(t_0, 1), 3), 4) \), resulting in the success condition \( \phi_3 \) holding in \( B \), and the associated declarative goal being no longer pursued (by replacing it with program nil). Finally, \( a_5 \) is chosen, adding its path predicate to \( B \) and asserting \( \neg p \). While it is now true that \( B \models \neg p \), HTN constraint \((4, p, 5)\) (and thus formula \( \neg \phi_3 \)) still holds, as \( p \) only needed to hold between \( \lambda_4 \) and \( a_5 \).

5. HTNs TO NON-PLANNING AGENTS

While the above approach ensures that a CANPlan agent will never make a choice that leads to the violation of an HTN constraint during execution, the approach does not suit traditional BDI agent programming languages such as CAN and AgentSpeak [17], which cannot perform any lookahead. For instance, given the plan-library in our running example, a CAN agent may well: (i) select plan-rule \( r_3 \) instead of \( r_3 \), thereby essentially violating HTN constraint \((4, p, 5)\) (when \( a_3 \) removes atom \( p \)); (ii) decompose \( \lambda_3 \) before executing action \( a_3 \), thereby essentially violating HTN constraint \((2 < 3)\); or (iii) substitute the variable \( Y \) that appears in \( r_1 \) with \( 2 \) instead of \( 1 \), thereby making \( r_4 \) the only applicable rule when decomposing \( \lambda_4 \).

Thus, in some sense, our translation produces CAN libraries that are ideally failure-free, i.e., failure-free in the “ideal case” where the agent begins execution in the “right” belief base and makes all the right choices during execution.

Definition 4 (Ideally Failure-Free). A (CAN) plan-library \( \Pi \) is ideally failure-free (relative to an action-library \( \Lambda \)) if for any rule \( e : \psi \leftarrow P \in \Pi \) and action sequence \( A \), there exists a belief base \( B \) and substitution \( \theta \) such that \( B \models \psi \theta \) (where \( \psi \theta \) is ground), and at least one trace in \( T(P, B, \Lambda, A, \Pi) \) is successful.

It is not difficult to see that a plan-library produced by our translation is indeed ideally failure-free. This follows from Theorem 2 and our assumption (in Section 3) that for any HTN domain \( D = (Op, Me) \) and method \( t \), there exists an initial state \( I \) such that \( \text{sol}(t, I, D) \neq \emptyset \).

Proposition 1. Let \( (Op, Me) \) be an HTN domain, and \( \Lambda_0 \) and \( \Pi_0 \) the corresponding CAN action- and plan-library, respectively. Then, \( \Pi_0 \) is ideally failure-free relative to \( \Lambda_0 \).

A more useful library for a CAN agent is one that is (always) failure-free. That is, if a plan-rule is applicable (its context condition holds in the current belief base), then the associated plan-body will not fail during execution (if there is no external interference) [12]. Thus, in some sense, the context conditions in such libraries endow an agent with certain “lookahead” abilities.

Definition 5 (Failure-Free). A (CAN) plan-library \( \Pi \) is said to be failure-free (relative to an action-library \( \Lambda \)) if for any plan-rule

![Figure 1: A CAN plan- and action-library depicted as a hierarchy. An arrow below a plan-rule indicates that its steps are ordered from left to right. Context conditions of plan-rules appear alongside them, and the effects of actions appear below them.](image-url)
(hvFC), which asserts that formal clothes were worn (wFC) and that casual clothes were not (¬wFC), or by wearing casual clothes if it is Friday. Travelling involves either walking if ¬wFC holds and the distance between X and Y is less than 2 miles (d(X, Y, 2)), or taking the bus if the agent has coins. Then, observe that by making the context condition of rule \( R_0 \) the following:

\[
\psi = ((\text{hvFC} \land \neg \text{friday}) \supset \text{hvCoins}) \land
\left((\neg \text{hvFC} \land \text{friday}) \supset (d(X, Y, 2) \lor \text{hvCoins})\right) \land
\left((\text{hvFC} \land \text{friday}) \supset \text{hvCoins}\right).
\]

the rule will only be applicable (and execute a first action) if no choice that might later be made during the plan-body’s execution, e.g. on reaching the (uncontrollable) choice between plan-rules \( R_1 \) and \( R_2 \), will result in the failure of a step.

Observe, however, that \( \psi \) above is “cautious”: it will not hold when the agent believes \( \text{hvFC} \land \text{friday} \land \neg \text{hvCoins} \land d(a, b, 2) \) (when pursuing \( \text{goToWork}(a,b) \)). Consequently, it will never pursue the execution (or HTN solution) where rule \( R_2 \) is chosen by chance, followed by \( R_3 \). More precisely, \( \{\Lambda, \Pi'\} \subseteq T_0(\Lambda, \Pi) \), where \( \Pi \) and \( \Lambda \) represent the libraries depicted in Figure 1, and \( \Pi' \) represents the modified one above. Importantly, such a loss in completeness cannot be avoided (while keeping the library failure-free) by simply rewriting \( \psi \) above. However, there are also failure-free CAN libraries with no such loss in completeness, e.g. the one depicted in Figure 1 but with the context conditions of \( R_1 \) and \( R_2 \) being mutually exclusive, i.e., \( \text{hvFC} \land \neg \text{friday} \) and \( \neg \text{hvFC} \land \text{friday} \), respectively. We can then remove the last conjunct in \( \psi \).

### 5.2 HTN Methods as BDI Plan-Rules

We shall now discuss a crucial restriction that must be imposed on HTNs in order to be able to convert HTN methods into CAN plan-rules in a way that makes the ordering explicit between event-goals and actions. This allows the (non-planning) CAN agent to follow “structural” guidelines regarding the execution order of such steps, and thereby also avoid the need to have “overly cautious” context conditions. For example, if the two event-goal programs in \( R_0 \) in Figure 1 are written not as sequential but as parallel ones (as in Section 4.1), the above context condition \( \psi \) of rule \( R_0 \) will need to be the “overly cautious” condition \( \psi = \text{false} \), to preclude the CAN agent from making (the uncontrollable) choice to travel before getting ready, which will essentially violate the associated HTN ordering constraint and lead to failure.

Analogously to the notion of a series-parallel graph [22], we call our restricted class of HTNs series-parallel HTNs. Intuitively, these are HTNs that can be incrementally constructed by the application of only sequential and parallel compositions of tasks.

**Definition 6 (Series-Parallel HTNs).** Let \( d_1 = [S_1, \phi_1] \) and \( d_2 = [S_2, \phi_2] \) be task networks that do not mention the same variables or task labels, and let the set of ordering constraints \( C = \{(n_1 < n_2) \mid (n_1 : t_1) \in S_1, (n_2 : t_2) \in S_2\} \).

- A **sequential product** of \( d_1 \) and \( d_2 \), denoted \( d_1 \circ d_2 \), is any task network \( d_3 = [S_1 \cup S_2, \phi_1 \land \phi_2 \land \phi_3 \land \bigwedge_{C \in C} \phi] \), where no ordering constraints occur in \( \phi_3 \).
- A **parallel product** of \( d_1 \) and \( d_2 \), denoted \( d_1 \parallel d_2 \), is any task network \( d_3 = [S_1 \cup S_2, \phi_1 \land \phi_2 \land \phi_3] \), where no ordering constraints occur in \( \phi_3 \).

Then, \( d = [S, \phi] \) is a **series-parallel** (SP) task network if either \( |S| = 1, d = d_1 \circ d_2 \), or \( d = d_1 \parallel d_2 \), for some SP task networks

\( d_1 \) and \( d_2 \). An HTN domain \((Op, Me)\) is a series-parallel one if for each method \([\tau, d] \in Me, d\) is an SP task network.

Figure 2 shows SP task networks and a non-SP task network depicted as DAGs. Observe that the latter cannot be constructed incrementally as defined above. For this to be possible, its task label 5 (resp. 6) must either be “inside” or “outside” the subgraph with vertices 2, 3, 4 and 6 (resp. 1, 2, 3 and 5), as in DAGs (c) and (f) (resp. (b) and (c)).

In some cases, HTN task networks can have implicit ordering constraints, which do not appear in the constraint formula. These must also be taken into account in order to “truly” recognise SP task networks. For example, if the negated between constraint \( \neg(n_1, p, n_2) \) (for some proposition \( p \)) occurs in a constraint formula, then ordering constraint \( n_1 < n_2 \) is “implicit” in it (if it does not occur in the formula), as the former requires a state to exist between \( n_1 \) and \( n_2 \) in which \( \neg p \) holds.

Moreover, unlike series-parallel graphs which are “static” in that they are orthogonal to any state, an implicit ordering constraint of an SP task network may only become “active” in certain states. For example, consider the primitive task network \( d = \{(n_1 : \tau_1), (n_2 : \tau_2)\}, \text{true} \), where \( \tau_1 \) asserts proposition \( p \), and \( \tau_1 \) and \( \tau_2 \) have preconditions \( \text{true} \) and \( p \), respectively. Then, any HTN solution (completion) for \( d \) will have (a ground instance of) \( \tau_1 \) before (a ground instance of) \( \tau_2 \) for initial states in which \( p \) does not hold. In other words, implicit constraint \( n_1 < n_2 \) only becomes “active” in such states. We therefore define implicit ordering constraints relative to a state.

**Definition 7 (Implicit Constraints).** Let \( d = [S, \phi] \) be a task network, \( I \) an initial state, \( D = (Op, Me) \) an HTN domain, and \( e = (n_1 < n_2) \) an ordering constraint that does not occur in \( \phi \), where \( n_1, n_2 \in S \). Then, \( e \) is an implicit ordering constraint of \( d \) relative to \( D \) and \( I \) if \( \text{sol}(d, I, D) = \text{sol}([S, \phi \land e], I, D) \).

An SP task network with implicit ordering constraints may in essence be a non-SP task network, and vice versa. For example, adding an edge from 5 to 3, and from 2 to 6 in the DAG in Figure 2(d) will essentially make it the DAG in Figure 2(a), which depicts a non-SP task network. Thus, we require below for implicit ordering constraints to be made explicit. The theorem states that some non-SP task networks cannot be captured by SP ones.

**Theorem 4.** There exists an initial state \( I \), a non-SP task network \( d = [S, \phi] \), and an HTN domain \( D \), such that \( \text{sol}(d, I, D) \neq \bigcup_{e \in [1,n]} \text{sol}(d_1, I, D) \) for any set of SP task networks \( d_1, \ldots, d_n \).

\(^7\text{Recall that } (n_1, p, n_2) \text{ holds vacuously if } n_2 \text{ precedes } n_1.\)
where each $d_i = [S, \phi_i]$, and $d_i$ and $d$ have no implicit ordering constraints (relative to $D$ and $T$).

Thus, in some situations, even multiple HTN methods containing SP task networks ($d_1, \ldots, d_n$) will not be able to yield the exact set of solutions as a single method containing a non-SP task network ($d$) with the same set of tasks.\footnote{An example of a situation in which this does not hold is when the DAGs in the figure depict primitive task networks.} To see why this holds, consider Figure 2. Suppose that each task label $i \in \{1, 6\}$ corresponds to a compound task that is decomposed into two primitive tasks, which we denote by $i^1$ and $i^2$. Then, the sequence of task labels

$1^1 \cdot 2^1 \cdot 3^1 \cdot 4^1 \cdot 5^1 \cdot 6^1 \cdot 3^2 \cdot 4^2 \cdot 5^2 \cdot 6^2 \cdot 4^1 \cdot 4^2$

represents an HTN solution for the non-SP task network in the figure, but not for any of the SP task networks, except for (d) which also produces “invalid” solutions: those that violate ordering constraints ($5 \prec 3$) and ($2 \times 6$) in (a).

Consequently, in some situations, a non-SP task network cannot be represented in terms of CAN plan-bodies encoded (purely) as sequential and parallel compositions of event-goals and actions (with no “implicit constraints”, e.g. in goals as in Section 4.2). However, any SP task network’s structure can be straightforwardly encoded as such a plan-body; e.g., Figure 2(b) represents the plan-body $\langle(5 \parallel (2 \; 6)) \; ; \; (3 \; 4), \; with \; the \; task \; labels \; here \; representing \; event-goals/actions. \; Conversely, \; any \; given \; goal-free \; CAN \; plan-body \; can \; also \; be \; translated \; into \; an \; SP \; task \; network.

**Proposition 2.** Let $\Pi$ and $A$ be BDI plan- and action-libraries, respectively, and $Me_{11}$ and $Op_A$ their corresponding HTN method- and operator-libraries, respectively, as defined in [20]. Then, $D = \langle Op_A, Me_{11} \rangle$ is an SP HTN domain.

The proof follows from Definition 6 and the translation provided in [20] from CAN libraries into HTN domains.

**Translating the Remaining HTN Constraints**

We conclude this section with a note regarding how the remaining HTN constraints could be translated into CAN entities.

Interestingly, we need to enforce a final restriction, namely, disallowing negated HTN ordering constraints. In particular, this is to disallow “forced” interleaving of steps. For example, constraint formula $\neg(n_1 \prec n_2) \land \neg(n_2 \prec n_1)$ forces all HTN solutions to have at least one primitive task associated with $n_1$ (or resp. $n_2$) occurring between two primitive tasks associated with $n_2$ (or resp. $n_1$). There is no corresponding construct in CAN to specify such “forced interleaving”.

Similarly, there is no CAN construct to check whether a literal $l$ holds “immediately before” a step $n$, to capture the HTN before constraint $(l, n)$. With some loss in generality, we could, however, capture this in CAN by checking that the literal holds “at some point before” the step, and then ensuring that the literal is never violated. For example, consider the before constraint $(p(X), n)$ and plan-body $a_1 \parallel a_2$, where action $a_1$ corresponds to $n$, and action $a_2$ asserts literal $\neg p(Y)$. Then, we first take plan-body $(\neg p(X) ; a_1) \parallel a_2$ (which checks that $p(X)$ holds at some point before $a_1$) and then change it into plan-body $(\neg p(X) ; a_1) ; a_2$, which ensures that $a_2$ is not interleaved between the test program and $a_1$, and thereby that $p(X)$ is not undone by $a_2$.\footnote{This is assuming that an agent’s top-level event-goals are not interleaved, or are interleaved with care [21]. In our example, we lose generality when variables $X$ and $Y$ are not substituted with the same constant: it is then acceptable for $a_2$ to occur before $a_1$.} HTN state constraints $(n, l)$ and $(n, l, n')$ can be translated similarly.

6. DISCUSSION AND FUTURE WORK

We could use existing algorithms for detecting such potential violations to CAN test programs, and avoiding them as described above. For example, in Partial Order Planning (POP), such resolutions are called “resolving threats” [25]. Similarly, while we have demonstrated that it is indeed possible to make CAN libraries failure-free (Section 5.1), it would be interesting to study how this could be automated. Some of the required techniques already exist, such as the ones used in [21, 7, 4] to “propagate” context conditions up the hierarchy and detect when they are potentially or definitely violated (e.g. in Figure 1. $R_3$’s context condition is potentially violated when $hvFC \land friday$ holds). We could ignore potentially violated context conditions when building the relevant conditionals for the higher-level “cautious” context condition (as we did when building the third conditional in $R_4$’s context condition).

Another avenue worth exploring is whether we could use extensions of the CAN operational semantics, or a different one altogether, to more closely capture certain aspects of HTNs. For example, in the extension of CAN to support “maintenance goals” [10], the violation of a “maintenance condition” is detected during BDI execution and an attempt is made to re-establish the condition via an available event-goal. Maintenance conditions might therefore serve as a useful representation for encoding HTN constraints.

In this paper we have demonstrated how HTN domains can be converted into corresponding BDI libraries. This allows for available HTN domains to be essentially “plugged in” to supplement a BDI agent’s plan-rules. When combined with lookahead-based execution, the new plan-rules are guaranteed to operate in accordance with the constraints specified by the designer of the HTN domain. To achieve this, we described a necessary alternative to the operational semantics of CANPlan, which now uses failure conditions in declarative goals as “control information” during lookahead. We also studied how HTN domains could be converted into the more traditional (AgentSpeak-like) CAN libraries, and the resulting loss in completeness. In particular, we characterised (i) a restricted class of HTN task networks that can be represented as “series-parallel” CAN-plan-bodies, and (ii) a useful class of CAN libraries called failure-free libraries, which, in some sense, have context conditions that “mimic” HTN-style lookahead. We showed how a failure-free library can always be obtained from any given plan-library.

7. ACKNOWLEDGEMENTS

The author is grateful to Lin Padgham and Sebastian Sardina for many useful discussions on closely related work while the author was a PhD student at RMIT University, and in particular for discussions relating to the insight in Section 5.1. The author would also like to thank the anonymous reviewers for their helpful feedback, and Brian Logan for pointing out that a translation from HTNs to BDI agent recipes might have useful applications.

REFERENCES


