BDI Agent Reasoning with Guidance from HTN Recipes

Lavindra de Silva
Institute for Advanced Manufacturing
University of Nottingham
Nottingham, UK
lavindra.desilva@nottingham.ac.uk

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 we 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

Appears in: *Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2017), S. Das, E. Durfee, K. Larson, M. Winikoff (eds.), May 8–12, 2017, São Paulo, Brazil.*

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* $\mathcal{B}$, i.e., a set of ground atoms; a *plan-library* Π, i.e., a set of *plan-rules*; and an *action-library* Λ. A *plan-rule* is of the form $e(\mathbf{t}){:}\psi \ \leftarrow P$, where $e(\mathbf{t})$ is an *event-goal*; $\mathbf{t}$ is a vector of terms; and ψ, a formula, is the *context condition*. The *plan-body* P is built from: *actions* $act(\mathbf{t})$; *belief addition* $+b$ and *removal* $-b$ programs which are used to respectively add and remove atom b from $\mathcal{B}$; *test programs* $?\phi$, where ϕ is a formula, which are used to test whether ϕ holds in $\mathcal{B}$; *event-goal programs* $!e$, where e is an event-goal; and *declarative goals* $\mathsf{Goal}(\phi_s, P, \phi_f)$, specifying that formula ϕ_s (the declarative goal) should be achieved using program P, failing if ϕ_f becomes true. As a plan-body "evolves", the following "internal" constructs may also be used: programs *nil*, $P_1 \rhd P_2$, 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 $P_1 \rhd P_2$ 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

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

The above grammar also describes a CANPlan plan-body when construct $\mathsf{Plan}(P)$ 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* $C \longrightarrow C'$ denotes that configuration C yields configuration C' in a single execution step. A *configuration* is a tuple $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P\rangle$, where $\mathcal{A}$ is the sequence of actions executed so far, and the other elements are as above; when comparing configurations, we ignore the static elements Π and Λ, and we sometimes omit them for brevity. CANPlan uses labelled transitions denoted $C \xrightarrow{\mathsf{lab}} C'$, where $\mathsf{lab} \in \{\mathsf{bdi}, \mathsf{plan}\}$, 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 $\langle \mathcal{B}, \mathcal{A}, !e\rangle$ evolves to $\langle \mathcal{B}, \mathcal{A}, (\!|\Delta|\!)\rangle$—with only an update to the last component—in one transition of type bdi or plan. Construct mgu stands for "most general unifier" [11], 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 $\mathsf{Plan}(P)$ necessitates one or more (internal) steps that yield a successful HTN decomposition of P.

$$\frac{\Delta = \{\psi_i\theta : P_i\theta \mid e' : \psi_i \leftarrow P_i \in \Pi \wedge \theta = \mathsf{mgu}(e, e')\}}{\langle \mathcal{B}, \mathcal{A}, !e\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, (\!|\Delta|\!)\rangle} \ D_1$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P'\rangle \quad \langle \mathcal{B}', \mathcal{A}', P'\rangle \xrightarrow{\mathsf{plan}_*} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle} \ D_2$$

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

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 *off-line* 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 $\langle d, \mathcal{I}, \mathcal{D}\rangle$, where d is a *task network* and $\mathcal{I}$ is an *initial state* (set of ground atoms). Element $\mathcal{D} = \langle Op, Me\rangle$ is an HTN (planning) *domain*, where *Op* is a set of STRIPS-like operators and *Me* is a set of *methods*. Elements $d, \mathcal{I}$, *Op* and *Me* are comparable to elements $P, \mathcal{B}, \Lambda$, and Π above (respectively).

A *task network* d is a tuple $[S, \phi]$, where S is a non-empty set of elements of the form $(n : \tau(\mathbf{t}))$; element n is a *task label*, which is unique in the planning problem; and $\tau(\mathbf{t})$ is either a *compound task* or *primitive task*. Element ϕ, 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 $(n \prec n')$ (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 τ 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 τ_1, τ_3 and τ_4.

$$\begin{aligned}
m_1 &= [\![\tau_1, [\{(2 : \tau_2(Y)), (3 : \tau_3)\}, 2 \prec 3 \wedge (q(Y), 2)]]\!] \\
m_2 &= [\![\tau_3, [\{(4 : \tau_4), (5 : \tau_5)\}, 4 \prec 5 \wedge (4, p, 5)]]\!] \\
m_3 &= [\![\tau_4, [\{(7 : \tau_7), (8 : \tau_8)\}, 7 \prec 8 \wedge (q(2), 7)]]\!] \\
m_4 &= [\![\tau_4, [\{(9 : \tau_9)\}, true]]\!]
\end{aligned}$$

Given an HTN planning problem $\langle d = [S_d, \phi_d], \mathcal{I}, \mathcal{D}\rangle$, with $\mathcal{D} = \langle Op, Me\rangle$, 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 τ and τ_m unify) to yield a new, and typically "more primitive" task network d'. Formally, this is denoted by $d' = reduce(d, n, m)$, and the set of all reductions of d is denoted by $red(d, \mathcal{D})$. Applying a reduction to $(n : \tau)$ above involves updating the set S_d by replacing $(n : \tau)$ with the tasks in d_m, and

CANPlan Entities	HTN Entities
belief base $\mathcal{B}$	state $\mathcal{I}$
action act (in plan-body)	primitive task
belief operations $+b$ and $-b$	primitive tasks
event-goal $!e$	compound task
test program $?\phi$	state constraints, e.g. (l, n)
plan-rule context condition ψ	state constraints
a sequence of two programs P ; P'	ordering constraints
programs in parallel $P \parallel P'$	no ordering constraints
plan-body P	task network $d = [S, \phi]$
plan-rule $e(\mathbf{t}) : \psi \leftarrow P$	method $[\![\tau, d]\!]$
plan-library Π	set of methods *Me*

Table 1: A summary of the core mapping from CANPlan to HTNs.

updating ϕ_d to include the constraints in d_m. Reductions are applied to the given task network until a *primitive task network* is obtained, i.e., one in which no compound tasks appear.

Finally, from a primitive task network, a *completion* σ is computed. Informally, this is a total ordering of a ground instance of the task network, that does not violate any constraints in the task network formula. Given elements d, $\mathcal{I}$ and $\mathcal{D}$ as above, the set of all completions is denoted as $comp(d, \mathcal{I}, \mathcal{D})$, and the set of all HTN (primitive) *solutions* as $sol(d, \mathcal{I}, \mathcal{D}) = \bigcup_{n<\omega} sol_n(d, \mathcal{I}, \mathcal{D})$, where $sol_n(d, \mathcal{I}, \mathcal{D})$ is defined inductively as

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

Intuitively, the set of HTN solutions for the planning problem is the set of all completions of all primitive task networks that can be obtained from zero or more reductions of d. Table 1 shows a summary from [19] of the conceptual mapping from CANPlan entities to HTN entities.

3. PRELIMINARIES

In this section we introduce some preliminary definitions and state the assumptions that we make. First, we define a "recovery-free" *execution trace* as a sequence of steps in which, if there is no external interference, there is no (BDI-style) recovery from a failed step (via program $P_1 \rhd P_2$). Failure occurs when a configuration $\langle \mathcal{B}, \mathcal{A}, P \neq nil \rangle$ is reached such that $\langle \mathcal{B}, \mathcal{A}, P \rangle \stackrel{\mathsf{plan}}{\not\longrightarrow}$ holds, namely, P has become "stuck". This will happen, for instance, if an event-goal program has no relevant plan-rules, resulting in the set Δ being empty in rule D_1 above. In the definition below, P_n denotes the last element in configuration C_n.

Definition 1 (Execution Trace). An *execution trace* of a configuration $C = \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, P \rangle$ is a finite sequence of configurations $C_1 \cdot \ldots \cdot C_n$ such that *(i)* $C = C_1$; *(ii)* either $P_n = nil$ or $C_n \stackrel{\mathsf{plan}}{\not\longrightarrow}$, i.e., the trace is "complete"; and *(iii)* $C_i \xrightarrow{\mathsf{plan}} C_{i+1}$ for all $i \in [1, n-1]$. If $P_n = nil$, then the trace is a *successful* execution trace, and it is a *failed* one otherwise. We use $\mathcal{T}(P, \mathcal{B}, \mathcal{A}, \Lambda, \Pi)$ to denote the set of all execution traces of C. ■

In [20], the authors propose one possible operational semantics for declarative goals, where failure conditions are ignored when goals are adopted from within a planning context. To this end, the authors introduce a new derivation rule, and adapt the original rule for goal adoption to make it applicable only in a non-planning context. The new rule (D_3) and the adapted one (D_4) are shown below.

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, !e, \phi_f) \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}, \mathcal{A}, (!e; ?\phi_s) \rangle} \; D_3$$

$$\frac{\mathcal{B} \not\models (\phi_s \vee \phi_f) \quad \langle \mathcal{B}, \mathcal{A}, !e \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P \rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, !e, \phi_f) \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Goal}(\phi_s, P \rhd P, \phi_f) \rangle} \; D_4$$

In rule D_4, program P is the original set of relevant plan-rules for event-goal program $!e$, 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 D_3, and remove the labels on D_4'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.

$$\frac{\langle \mathcal{B}, \mathcal{A}, P \rangle \stackrel{\mathsf{bdi}}{\not\longrightarrow} \quad \langle \mathcal{B}, \mathcal{A}, P' \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P'' \rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Goal}(\phi_s, P \rhd P', \phi_f) \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Goal}(\phi_s, P'' \rhd P', \phi_f) \rangle}$$

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 $\mathcal{D} = \langle Op, Me \rangle$, for any method $[\![\tau, d]\!] \in Me$, there exists an initial state $\mathcal{I}$ such that $sol(d, \mathcal{I}, \mathcal{D}) \neq \emptyset$. 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)* $\phi \supset (n \prec n')$ 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 ground 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. To this end, the main problem that needs to be addressed is how to convert HTN methods into equivalent CANPlan plan-rules. 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 ϕ is a conjunction of possibly negated HTN constraints. An HTN *domain* $\langle Op, Me \rangle$ 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

[1] When 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 $\mathcal{D} = \langle Op, Me \rangle$ be an HTN domain. There exists a conjunctive HTN domain $\mathcal{D}' = \langle Op, Me' \rangle$ such that for any labelled task $(n : \tau)$ and initial state $\mathcal{I}$, $sol(d, \mathcal{I}, \mathcal{D}) = sol(d, \mathcal{I}, \mathcal{D}')$, where $d = [\{(n : \tau)\}, true]$. ■

Given an HTN domain $\mathcal{D}$ as above, we obtain a conjunctive domain $\mathcal{D}' = \langle Op, Me' \rangle$ as follows. For each method $[\![\tau, [S, \phi]]\!] \in Me$, we add the method $[\![\tau, [S, \phi']]\!]$ to Me' (which is initially empty) for each disjunct ϕ' appearing in ϕ_{dnf}, where ϕ_{dnf} is ϕ 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 $\mathcal{D} = \langle Op, Me \rangle$ an HTN domain. If there exists a method $[\![\tau', [S, \phi]]\!] \in Me$ such that τ' and τ 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 $\mathcal{D}$, denoted $\text{FIN}((n : \tau), \mathcal{D})$, is defined inductively as

$$S_1 \cup \bigcup_{(n'':\tau'') \in S_2} \text{FIN}((n'' : \tau''), \mathcal{D}).$$

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

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

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 $\mathbf{f}$. Formally, a *path function* (or *path*) is defined inductively as the 2-ary (FOL) function symbol $\mathbf{f}(t_1, t_2)$, where t_2 is a variable or task label, and t_1 is either the distinguished constant top or a path function. For example, if task label n_3 corresponds to an action, then $\mathbf{f}(\mathbf{f}(\mathbf{f}(top, n_1), n_2), n_3)$ represents two decompositions of the top-level event-goal (compound task) corresponding to label n_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 $\mathcal{B}$.[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 ϕ follows from a belief base $\mathcal{B}$ [20]. Thus, we assume that an operation $\mathcal{B} \models_A \phi$ is provided, where $A = \{\mathbf{A1}, \mathbf{A2}, \mathbf{A3}\}$ is our set of axioms.

Axiom (Prefix & End).

$$\begin{array}{llll} (\mathbf{A1}) & \text{PREF}(X,Y) & \leftarrow & X = Y \\ (\mathbf{A2}) & \text{PREF}(X,Y) & \leftarrow & \exists R, S\big(Y = \mathbf{f}(R,S) \wedge \text{PREF}(X,R)\big) \\ (\mathbf{A3}) & \text{END}(X,Y) & \leftarrow & \exists R\big(Y = \mathbf{f}(R,X)\big) \end{array}$$

Finally, by including predicate $\mathbf{p}(Z)$ in the add-list (Φ^+) of every action-rule in the agent's action-library Λ, where $\mathbf{p}$ is a distinguished predicate and Z is a variable that will always bind to the path that led to the action, the following condition can be used to query the belief base $\mathcal{B}$—while a plan-body is being executed—to check whether a particular event-goal occurring in the body has executed its first action:

$$\Phi^1(\mathbf{f}, \mathcal{D}) \overset{\text{def}}{=} \exists Y, \mathbf{p}(Y) \wedge \text{PREF}(\mathbf{f}, Y),$$

where $\mathbf{f} = \mathbf{f}(X, n)$, and intuitively: variable X represents the path that led to the event-goal, n corresponds to the event-goal itself, and Y represents the path that led to an action that was executed. (We shall make these clearer in Section 4.2.) Similarly, the following condition can be used to check whether the event-goal has executed all of its associated actions:

$$\Phi^+(\mathbf{f}, \mathcal{D}) \overset{\text{def}}{=} \bigvee_{S \in \text{FIN}^*((n:\tau),\mathcal{D})} \bigwedge_{(n':\tau') \in S} \exists Y, \mathbf{p}(Y) \wedge \text{PREF}(\mathbf{f}, Y) \wedge \text{END}(n', Y),$$

where $\mathbf{f}$ is as above. Informally, it is sufficient to check whether one set of terminating tasks of the event-goal has finished execution.

We now have the necessary formal machinery for translating a given HTN domain and planning problem into their BDI counterparts. Since an HTN initial state and operator-library have similar representations to a CAN belief base and action-library, respectively, we shall restrict our attention to translating HTN methods.

4.1 HTN Methods as BDI Plan-Rules

Given an HTN method $m = [\![\tau(\mathbf{t}), [S, \phi_{htn}]]\!] \in Me$, the corresponding CAN plan-rule is $e_m : \psi_m \leftarrow P_m$, where each of these components is defined as follows.

The event-goal. We take $e_m = \tau(\mathbf{t} \cdot X)$, namely, we use the corresponding HTN compound task and append X to its vector of terms $\mathbf{t}$. The former is a variable that will always bind to the path that led to the event-goal.

The context condition. Let task $(n_1 : \tau_1) \in S$ be the one that precedes all the other tasks in S. Recall that such a task always exists due to our assumption in Section 3. Then ψ_m is the conjunction of the literals in the following set:

$$\{l \mid (l, n_1) \text{ occurs in } \phi_{htn}\} \cup \{\neg l \mid \neg(l, n_1) \text{ occurs in } \phi_{htn}\}.$$

Intuitively, the context condition is composed of the literals that need to hold before the first task in the task network.

The plan-body. Given the variable X above and the HTN method's set of tasks $S = \{(n_1 : \tau_1), \ldots, (n_m : \tau_m)\}$, for each $(n_i : \tau_i(\mathbf{t}_i))$ we either create the event-goal program $P_i = !\tau_i(\mathbf{t}'_i)$ (if τ_i is compound) or the action $P_i = \tau_i(\mathbf{t}'_i)$ (if τ_i is primitive), where $\mathbf{t}'_i = \mathbf{t}_i \cdot \mathbf{f}(X, n_i)$. Then, we take the plan-body P_m above as the declarative-goal

$$\mathsf{Goal}(\mathbf{p}(X), P, \phi_f),$$

where program

$$P = \big(P_1 \parallel \ldots \parallel P_m\big); +\mathbf{p}(X)$$

with each P_i as defined above. Informally, the declarative goal can be accomplished if $\mathbf{p}(X)$ 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 ϕ_f hold. Intuitively, combining this declarative goal with the lookahead construct (Plan) allows "monitoring" the program P being pursued and "backtracking" when a step makes ϕ_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]

$$\begin{array}{lcl} r_1 & = & !e_1 : q(Y) \leftarrow \mathsf{Goal}(\phi_{s_1}, P_1, \phi_{f_1}) \\ r_2 & = & !e_3 : \mathit{true} \leftarrow \mathsf{Goal}(\phi_{s_2}, P_2, \phi_{f_2}) \\ r_3 & = & !e_4 : q(2) \leftarrow \mathsf{Goal}(\phi_{s_3}, P_3, \phi_{f_3}) \\ r_4 & = & !e_4 : \mathit{true} \leftarrow \mathsf{Goal}(\phi_{s_4}, (a_9; +\phi_{s_4}), \mathit{false}) \end{array}$$

where

$$\begin{array}{lcl} P_1 & = & (a_2(Y) \parallel !e_3); +\phi_{s_1} \\ P_2 & = & (!e_4 \parallel a_5); +\phi_{s_2} \\ P_3 & = & (a_7 \parallel a_8); +\phi_{s_3} \end{array}$$

4.2 HTN Constraints as Failure Conditions

We take the failure condition ϕ_f as the negation of the formula $\textsc{trans}(c_1) \wedge \ldots \wedge \textsc{trans}(c_k)$, where each c_i is a conjunct in the HTN constraint formula ϕ_{htn} above, and $\textsc{trans}(c_i)$ is defined below. In what follows, we ignore the HTN domain $\mathcal{D}$; e.g. we write $\Phi^1(\mathbf{f})$ instead of $\Phi^1(\mathbf{f}, \mathcal{D})$. Let us now consider the possible values for a conjunct c in ϕ_{htn}.

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

  $$\Phi^1(\mathbf{f}_{n_2}) \models \Phi^+(\mathbf{f}_{n_1}),$$

  where $\mathbf{f}_i$ denotes $\mathbf{f}(X, i)$. 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, $\textsc{trans}(c)$ 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:

  $$\big(\Phi^+(\mathbf{f}_{n_1}) \models l\big) \vee \Phi^1(\mathbf{f}_{n_2}).$$

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

  $$\big(\Phi^1(\mathbf{f}_{n'}) \models l\big) \vee \Phi^1(\mathbf{f}_n).$$

  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 $c = (n, l)$, 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 $\textsc{trans}(c)$ as respectively $\textsc{trans}((\neg l, n))$ or $\textsc{trans}((n, \neg l))$. If $c = \neg(n_1 \prec n_2)$, then $\textsc{trans}(c)$ is the converse of the formula corresponding to the case where $c = (n_1 \prec n_2)$, 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.[5] For example, the failure conditions ϕ_{f_1}, ϕ_{f_2}, and ϕ_{f_3} in the declarative goals of our running example are the negations of the following formulas, respectively:

$$\Phi^1(\mathbf{f}_3) \models \Phi^+(\mathbf{f}_2), \quad \big(\Phi^+(\mathbf{f}_4) \models p\big) \vee \Phi^1(\mathbf{f}_5), \quad \Phi^1(\mathbf{f}_8) \models \Phi^+(\mathbf{f}_7).$$

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 $(n : \tau(\mathbf{t}))$ if and only if the corresponding CANPlan agent finds the same solution by performing lookahead on the corresponding event-goal program $!\tau(\mathbf{t} \cdot \mathbf{f}(top, n))$, which we denote as $!e_n$. In the theorem below, subscripts denote the HTN entities that were used to obtain the corresponding CANPlan entities; e.g., Π_{Me} is the CANPlan plan-library obtained from the HTN method-library *Me*, via the translation above.

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

$$\sigma \in sol(d, \mathcal{I}, \mathcal{D}) \text{ iff for some } \mathcal{B}'$$
$$\langle \Pi_{Me}, \Lambda_{Op}, \mathcal{B}_{\mathcal{I}}, \mathcal{A}, \mathsf{Plan}(!e_n) \rangle \xrightarrow{\mathsf{bdi}_*} \langle \Pi_{Me}, \Lambda_{Op}, \mathcal{B}', \mathcal{A} \cdot \sigma, nil \rangle.$$

■

Thus, any sequence of actions σ executed by the agent from belief base $\mathcal{B}_{\mathcal{I}}$, in order to achieve event-goal $!e_n$ via the libraries Π_{Me} and Λ_{Op} (translated from $\mathcal{D}$), will be an HTN solution for the corresponding HTN planning problem, and vice versa.[6] 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 $C_1 \cdot \ldots \cdot C_k$, such that for each C_i, no adopted (and possibly "nested") declarative goal appearing in P_i has a failure condition that holds in $\mathcal{B}_i$ (where P_i and $\mathcal{B}_i$ 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 τ_5 and τ_9 remove atom p.

Consider a possible lookahead-based execution of event-goal program $!e_1$, i.e., a particular execution trace of the CANPlan program $\mathsf{Plan}(!e_1(\mathbf{t} \cdot \mathbf{f}(top, 1)))$, from a belief base $\mathcal{B}_0 \models q(1) \wedge q(2)$. First, plan-rule r_1 is selected to achieve $!e_1$, 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 ϕ_{s_1} nor the failure condition ϕ_{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 $a_2(1)$ is chosen (arbitrarily) from the parallel steps in P_1, resulting in atom $q(1)$ being removed from $\mathcal{B}_0$, and atom p

[4] For legibility, we ignore constraint $4 \prec 5$, some path functions in plan-rules, and HTN before constraints.

[5] The case where $c = \neg(n_1, l, n_2)$—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.

[6] 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" $\mathbf{p}(\mathbf{f}(\mathbf{f}(top, 1), 2))$ being added to it, yielding the updated belief base $\mathcal{B}$. Following this, plan-rule r_2 is selected to achieve event-goal program $!e_3$, and plan-rule r_3 is selected to achieve $!e_4$, resulting in their associated declarative goals also being adopted.

In our scenario, the next action that is chosen is a_7, adding atom $\mathbf{p}(\mathbf{f}(\mathbf{f}(\mathbf{f}(\mathbf{f}(top, 1), 3), 4), 7))$ to $\mathcal{B}$. At this point, $\mathcal{B}$ still does not entail ϕ_{f_1}: while $!e_3$ did indeed just execute its first action, this was done only after a_2. Similarly, action a_8 is chosen next, adding its associated path predicate to $\mathcal{B}$. At this point, $\mathcal{B}$ still does not entail ϕ_{f_3}: action a_8 was indeed just executed, but only after a_7. Moreover, $\mathcal{B}$ still does not entail ϕ_{f_2} either: all actions associated with $!e_4$ were just executed, but proposition p also holds in $\mathcal{B}$ (due to a_2). The belief addition $+\phi_{s_3}$ is then executed with $\phi_{s_3} = \mathbf{p}(\mathbf{f}(\mathbf{f}(\mathbf{f}(top, 1), 3), 4))$, resulting in the success condition ϕ_{s_3} holding in $\mathcal{B}$, 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 $\mathcal{B}$ and asserting $\neg p$. While it is now true that $\mathcal{B} \models \neg p$, HTN constraint $(4, p, 5)$ (and thus formula $\neg\phi_{f_2}$) still holds, as p only needed to hold *between* $!e_4$ 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_4 instead of r_3, thereby essentially violating HTN constraint $(4, p, 5)$ (when a_9 removes atom p); *(ii)* decompose $!e_3$ before executing action a_2, thereby essentially violating HTN constraint $(2 \prec 3)$; 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 $!e_4$.

Thus, in some sense, our translation produces CAN libraries that are *ideally failure-free*, i.e., failure-free in the "ideal" cases 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 Π is *ideally failure-free* (relative to an action-library Λ) if for any rule $e : \psi \leftarrow P \in \Pi$ and action sequence $\mathcal{A}$, there exists a belief base $\mathcal{B}$ and substitution θ such that $\mathcal{B} \models \psi\theta$ (where $\psi\theta$ is ground), and at least one trace in $\mathcal{T}(P\theta, \mathcal{B}, \mathcal{A}, \Lambda, \Pi)$ 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 $\mathcal{D} = \langle Op, Me \rangle$ and method $[\![\tau, d]\!] \in Me$, there exists an initial state $\mathcal{I}$ such that $sol(d, \mathcal{I}, \mathcal{D}) \neq \emptyset$.

Proposition 1. Let $\langle Op, Me \rangle$ be an HTN domain, and Λ_{Op} and Π_{Me} the corresponding CAN action- and plan-library, respectively. Then, Π_{Me} is ideally failure-free relative to Λ_{Op}. ■

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 Π is said to be *failure-free* (relative to an action-library Λ) if for any plan-rule $e : \psi \leftarrow P \in \Pi$, ground instance $e\theta$ of e, belief base $\mathcal{B}$, and action sequence $\mathcal{A}$, if $\mathcal{B} \models \psi\theta\theta'$ (where $\psi\theta\theta'$ is ground), any trace in $\mathcal{T}(P\theta\theta', \mathcal{B}, \mathcal{A}, \Lambda, \Pi)$ is successful. ■

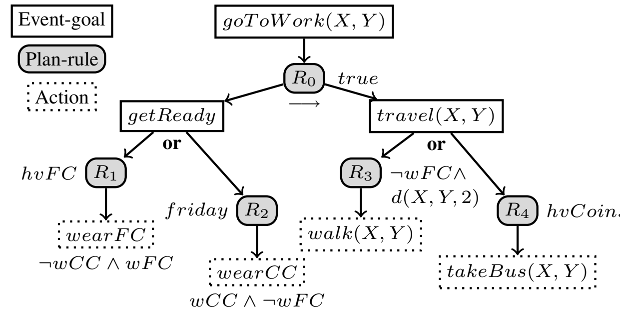

Figure 1: A CAN plan- and action-library depicted as a hierarchy. An arrow below a plan-rule indicates that its steps are ordered from left to right. Context conditions of plan-rules appear alongside them, and the effects of actions appear below them.

In order to obtain such a CAN library, context conditions need to be made sufficiently "cautious", and doing so involves determining (offline) which choices the agent will or may make under different situations, and the resulting violations to context conditions. To minimise the uncertainty regarding such choices, and consequently avoid the need for "overly cautious" context conditions, HTN "control knowledge" in the form of ordering constraints should be made explicit in BDI plan-bodies. This contrasts with our approach in the previous section, where such knowledge was implicitly encoded in failure conditions. In the next two sections, we explore the following two points, as well as their associated losses in completeness: *(i)* how a CAN plan-library can be made failure-free; and *(ii)* how an HTN method can be converted into a CAN plan-rule in which HTN ordering constraints are represented as sequential and parallel compositions of event-goal programs and actions.

5.1 Making CAN Libraries Failure-Free

It turns out that we can always obtain a failure-free CAN library from a given plan-library (such as one that was translated from an HTN domain), by adding "guards" into context conditions. To state this formally, we introduce the following auxiliary notions. First, given only a program P, an action-library Λ, and a plan-library Π, we use $\mathcal{T}(P, \Lambda, \Pi)$ to denote the union of all sets $\mathcal{T}(P, \mathcal{B}, \mathcal{A}, \Lambda, \Pi)$ for any $\mathcal{B}$ and $\mathcal{A}$. Second, given only an action-library Λ and a plan-library Π, we use $\mathcal{T}(\Lambda, \Pi)$ to denote the union of all sets $\mathcal{T}(P\theta\theta', \mathcal{B}, \mathcal{A}, \Lambda, \Pi)$, for any $\mathcal{B}$, $\mathcal{A}$, and ground plan-body $P\theta\theta'$ taken from the library, i.e., where a rule $e : \psi \leftarrow P \in \Pi$ exists with $\mathcal{B} \models \psi\theta$. Finally, $\mathcal{T}_S(\Lambda, \Pi)$ is the set of all successful execution traces in $\mathcal{T}(\Lambda, \Pi)$, i.e., those that end in a configuration in which the plan-body is *nil*.

Theorem 3. Let Π be a plan-library and Λ an action-library. There exists a failure-free plan-library Π' that can be obtained from Π by replacing each rule $e : \psi \leftarrow P \in \Pi$ with some rule $e : \psi \wedge \psi' \leftarrow P$ (possibly with $\psi' = true$). ■

Observe that for any such Π', we have $\mathcal{T}(\Lambda, \Pi') \subseteq \mathcal{T}_S(\Lambda, \Pi)$. The theorem holds because we can, in extreme cases, take one or more of the guards ψ' above as *false*. As an example of what these guards might look like in general, consider Figure 1. This depicts a CAN plan- and action-library for going to work from a location X to Y, which involves getting ready and then travelling. The former is either achieved by wearing formal clothes if the agent has them

($hvFC$), which asserts that formal clothes were worn (wFC) and that casual clothes were not ($\neg wCC$), or by wearing casual clothes if it is Friday. Travelling involves either walking if $\neg wFC$ 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 $\mathcal{R}_0$ the following:

$$\begin{aligned}\psi \;=\; & \big((hvFC \land \neg friday) \supset hvCoins\big) \;\land \\ & \big((\neg hvFC \land friday) \supset (d(X,Y,2) \lor hvCoins)\big) \;\land \\ & \big((hvFC \land friday) \supset hvCoins\big),\end{aligned}$$

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 ψ above is "cautious": it will not hold when the agent believes $hvFC \land friday \land \neg hvCoins \land d(a,b,2)$ (when pursuing $!goToWork(a,b)$). Consequently, it will never pursue the execution (or HTN solution) where rule R_2 is chosen by chance, followed by R_3. More precisely, $\mathcal{T}(\Lambda,\Pi') \subset \mathcal{T}_S(\Lambda,\Pi)$, where Π and Λ represent the libraries depicted in Figure 1, and Π' represents the modified one above. Importantly, such a loss in completeness cannot be avoided (while keeping the library failure-free) by simply rewriting ψ 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., $hvFC \land \neg friday$ and $\neg hvFC \land friday$, respectively. We can then remove the last conjunct in ψ.

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 steps but as parallel ones (as in Section 4.1), the above context condition ψ of rule R_0 will need to be the "overly cautious" condition $\psi = \mathit{false}$, 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 $C = \{(n_1 \prec n_2) \mid (n_1 : t_1) \in S_1, (n_2 : t_2) \in S_2\}$.

- A ***sequential product*** of d_1 and d_2, denoted $d_1 \circ d_2$, is any task network $d_3 = [S_1 \cup S_2, \phi_1 \land \phi_2 \land \phi_3 \land \bigwedge_{c\in C} c]$, where no ordering constraints occur in ϕ_3.[7]
- A ***parallel product*** of d_1 and d_2, denoted $d_1 \mid d_2$, 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 ϕ_3.

Then, $d = [S, \phi]$ is a ***series-parallel*** (SP) task network if either $|S| = 1$, $d = d_1 \circ d_2$, or $d = d_1 \mid d_2$, for some SP task networks d_1 and d_2. An HTN domain $\langle Op, Me\rangle$ is a ***series-parallel*** one if for each method $[\![\tau, d]\!] \in Me$, d is an SP task network. ■

[7] Strictly speaking, one or more "transitive" ordering constraints can also be removed from the formula in d_3.

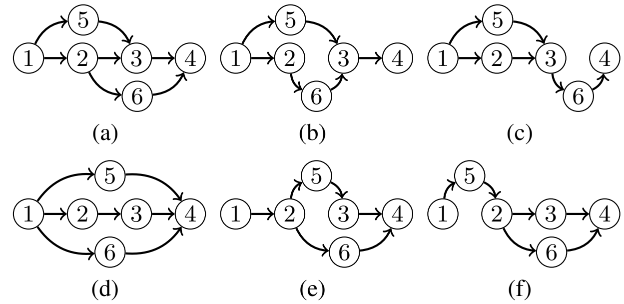

Figure 2: DAGs depicting the structures of a non-SP task network (a) and SP task networks (b)-(f). Vertices represent HTN task labels, and an edge (i,j) represents the ordering constraint $(i \prec j)$.

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 (e) 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 $(n_1 \prec n_2)$ 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.[8]

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 $d = [\{(n_1 : \tau_1), (n_2 : \tau_2)\}, true]$, where τ_1 asserts proposition p, and τ_1 and τ_2 have preconditions $true$ and p, respectively. Then, any HTN solution (completion) for d will have (a ground instance of) τ_1 before (a ground instance of) τ_2 for initial states in which p does not hold. In other words, implicit constraint $(n_1 \prec n_2)$ 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, $\mathcal{I}$ an initial state, $\mathcal{D} = \langle Op, Me\rangle$ an HTN domain, and $c = (n_1 \prec n_2)$ an ordering constraint that does not occur in ϕ, where $n_1, n_2 \in S$. Then, c is an ***implicit ordering constraint*** of d relative to $\mathcal{D}$ and $\mathcal{I}$ if $sol(d,\mathcal{I},\mathcal{D}) = sol([S, \phi \land c],\mathcal{I},\mathcal{D})$. ■

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 $\mathcal{I}$, a non-SP task network $d = [S, \phi]$, and an HTN domain $\mathcal{D}$, such that $sol(d,\mathcal{I},\mathcal{D}) \neq \bigcup_{i\in[1,n]} sol(d_i,\mathcal{I},\mathcal{D})$ for any set of SP task networks $d_1, \ldots, d_n$,

[8] Recall that (n_1, p, n_2) holds vacuously if n_2 precedes n_1.

where each $d_i = [S, \phi_i]$, and d_i and d have no implicit ordering constraints (relative to $\mathcal{D}$ and $\mathcal{I}$). ■

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.[9] 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

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

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 $(2 \prec 6)$ 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 $1 \; ; \; (5 \parallel (2 \; ; \; 6)) \; ; \; 3 \; ; \; 4$, 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 Π and Λ be BDI plan- and action-libraries, respectively, and Me_Π and Op_Λ their corresponding HTN method- and operator-libraries, respectively, as defined in [20]. Then, $\mathcal{D} = \langle Op_\Lambda, Me_\Pi \rangle$ 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 $\neg(n_1 \prec n_2) \wedge \neg(n_2 \prec n_1)$ 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 $(?p(X) \; ; \; a_1) \parallel a_2$ (which checks that $p(X)$ holds at some point before a_1) and then change it into plan-body $(?p(X) \; ; \; a_1) \; ; \; a_2$, 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.[10] HTN state constraints (n, l) and (n, l, n') can be translated similarly.

[9] An example of a situation in which this does *not* hold is when the DAGs in the figure depict primitive task networks.

[10] 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.

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 $hvFC \wedge friday$ 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_0'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

[1] N. Alechina, R. H. Bordini, J. F. Hübner, M. Jago, and B. Logan. Belief revision for AgentSpeak agents. In *Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1288–1290, 2006.

[2] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents: Components for intelligent agents in Java. *AgentLink Newsletter*, 2:2–5, Jan. 1999. Agent Oriented Software Pty. Ltd.

[3] B. J. Clement and E. H. Durfee. Exploiting domain knowledge with a concurrent hierarchical planner. In *Proceedings of the Workshop on Analysing and Exploiting*

Domain Knowledge for Efficient Planning, Working Notes, pages 57–62, 2000.

[4] B. J. Clement, E. H. Durfee, and A. C. Barrett. Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research (JAIR)*, 28:453–515, 2007.

[5] A. Coles, A. Coles, A. G. Olaya, S. Jiménez, C. L. López, S. Sanner, and S. Yoon. A survey of the seventh international planning competition. *Artificial Intelligence Magazine*, 33(1):83–88, 2012.

[6] L. de Silva and L. Padgham. A comparison of BDI based real-time reasoning and HTN based planning. In *Proceedings of the Australian Joint Conf. on Artificial Intelligence*, pages 1167–1173, 2004.

[7] L. de Silva, S. Sardina, and L. Padgham. Summary information for reasoning about hierarchical plans. In *European Conf. on Artificial Intelligence (ECAI)*, pages 1300–1308, 2016.

[8] K. Erol, J. A. Hendler, and D. S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the National Conf. on Artificial Intelligence (AAAI)*, pages 1123–1228, 1994.

[9] J. R. Firby. *Adaptive Execution in Complex Dynamic Domains*. PhD thesis, Yale University, 1989. Technical Report YALEU/CSD/RR #672.

[10] J. Harland, D. N. Morley, J. Thangarajah, and N. Yorke-Smith. An operational semantics for the goal life-cycle in BDI agents. *Autonomous Agents and Multi-Agent Systems*, 28(4):682–719, 2014.

[11] J. W. Lloyd. *Foundations of Logic Programming*. Springer, second edition, 1987.

[12] F. Meneguzzi and M. Luck. Leveraging new plans in AgentSpeak(PL). In *Proceedings of the International Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 5397 of *Lecture Notes in Computer Science (LNCS)*, pages 111–127, 2009.

[13] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, D. Wu, F. Yaman, H. Muñoz-Avila, and W. Murdock. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2):34–41, 2005.

[14] D. S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 968–973, 1999.

[15] M. Paolucci, O. Shehory, K. P. Sycara, D. Kalp, and A. Pannu. A planning component for RETSINA agents. In *Agent Theories, Architectures, and Languages*, pages 147–161, 1999.

[16] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, 1981.

[17] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the European Workshop on Modeling Autonomous Agents in a Multi-Agent World (Agents Breaking Away)*, volume 1038 of *Lecture Notes in Computer Science (LNCS)*, pages 42–55, 1996.

[18] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proceedings of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 473–484, 1991.

[19] S. Sardina, L. de Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proceedings of the Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1001–1008, 2006.

[20] S. Sardina and L. Padgham. A BDI agent programming language with failure recovery, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, 2011.

[21] J. Thangarajah and L. Padgham. Computationally effective reasoning about goal interactions. *Journal of Automated Reasoning*, 47(1):17–56, 2011.

[22] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 1–12, 1979.

[23] D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):197–227, 1995.

[24] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proceedings of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 470–481, 2002.

[25] H. L. S. Younes and R. G. Simmons. VHPOP: versatile heuristic partial order planner. *Journal of Artificial Intelligence Research (JAIR)*, 20:405–430, 2003.