HTN Acting: A Formalism and an Algorithm

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

ABSTRACT

Hierarchical Task Network (HTN) planning is a practical and efficient approach to planning when the 'standard operating procedures' for a domain are available. Like Belief-Desire-Intention (BDI) agent reasoning, HTN planning performs hierarchical and context-based refinement of goals into subgoals and basic actions. However, while HTN planners 'lookahead' over the consequences of choosing one refinement over another, BDI agents interleave refinement with acting. There has been renewed interest in making HTN planners behave more like BDI agent systems, e.g. to have a unified representation for acting and planning. However, past work on the subject has remained informal or implementation-focused. This paper is a formal account of 'HTN acting', which supports interleaved deliberation, acting, and failure recovery. We use the syntax of the most general HTN planning formalism and build on its core semantics, and we provide an algorithm which combines our new formalism with the processing of exogenous events. We also study the properties of HTN acting and its relation to HTN planning.

KEYWORDS

HTN Acting; HTN Planning; BDI Agent Programming Languages

ACM Reference Format:
Lavindra de Silva. 2018. HTN Acting: A Formalism and an Algorithm. In *Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), Stockholm, Sweden, July 10–15, 2018,* IFAAMAS, 9 pages.

1 INTRODUCTION

Hierarchical Task Network (HTN) planning [8, 12, 17, 18] is a practical and efficient approach to planning when the 'standard operating procedures' for a domain are available. HTN planning is similar to Belief-Desire-Intention (BDI) [13, 19, 20, 27] agent reasoning in that both approaches perform hierarchical and context-based refinement of goals into subgoals and basic actions [21, 22]. However, while HTN planners 'lookahead' over the consequences of choosing one refinement over another before suggesting an action, BDI agents interleave refinement with acting in the environment. Thus, while the former approach can guarantee goal achievability (if there is no action failure or environmental interference), the latter approach is able to quickly respond to environmental changes and exogenous events, and recover from failure. This paper presents a formal semantics that builds on the core HTN semantics in order to enable such response and recovery.

Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), M. Dastani, G. Sukthankar, E. André, S. Koenig (eds.), July 10–15, 2018, Stockholm, Sweden. © 2018 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

One motivation for our work is a recent drive toward adapting the languages and algorithms used in Automated Planning to build a framework for 'refinement acting' [11], i.e., deciding how to carry out a chosen recipe of action to achieve some objective, while dealing with environmental changes, events, and failures. To this end, [11] proposes the Refinement Acting Engine (RAE), an HTN-like framework with continual online processing and recipe repair in the case of runtime failure. A key consideration in the RAE is a unified hierarchical representation and a core semantics that suits the needs of both acting and lookahead. We are also motivated by recent work [4] which suggests that a fragment of the recipe language of HTN planning does not have a direct (nor known) translation to the recipe languages of typical BDI agent programming languages such as AgentSpeak [19] and CAN [27]. For example, HTNs allow a flexible specification of how steps in a recipe should be interleaved, whereas steps in CAN recipes must be sequential or interleaved in a 'series-parallel' [25] manner.

There have already been some efforts toward adapting HTN planning systems to make them behave more like BDI agent systems. Perhaps the first of these efforts was the RETSINA architecture [24], which used an HTN language and semantics for representing recipes and refining tasks, but also interleaved task refinement with acting in the environment. RETSINA is an implemented architecture which has been used in a range of real-world applications. In [5], the JSHOP [17] HTN planner is modified in two ways: *(i)* to execute a solution (comprising a sequence of actions) found via lookahead, and then re-plan if the solution is no longer viable in the real world (due to a change in the environment), and *(ii)* to immediately execute the chosen refinement for a task, instead of first performing lookahead to check whether the refinement will accomplish the task. The latter modification made JSHOP as effective as the industry-strength JACK BDI agent framework [26], in terms of responsiveness to environmental changes.

However, both RETSINA and the JSHOP variant lack a formalism, making it difficult to study the properties (e.g. correctness) of their semantics, and to compare them to other similar systems. The same applies to the algorithms and abstract syntax of the RAE framework, which are presented only in pseudocode.

There is also some work on making BDI-like agent systems behave more like HTN planning systems. In particular, both the REAP algorithm in [11] and the CANPlan [21, 22] BDI agent programming language can make informed decisions about refinement choices by using a lookahead capability. Similarly, there are agent programming languages and systems that support some form of planning (though not HTN-style planning) [16], such as the PRS [10] based Propice-Plan [6] system and the situation-calculus based IndiGolog [3] system. Finally, there are also some interesting extensions to HTN and HTN-like planning [1, 2, 9, 14, 23, 28], e.g. approaches that combine classical and HTN planning. In contrast, our work is not concerned with lookahead or planning, but with adapting the HTN planning semantics to enable BDI-style behaviour.

Thus, our contribution is a formal account of HTN *acting*, which supports interleaved deliberation, acting, and recovery from failure, e.g. due to environmental changes. To this end, we use the syntax of the most general HTN planning formalism [7, 8], and we build on its core semantics by developing three main definitions: execution via reduction, action, and replacement. We then provide an algorithm for HTN acting which combines our new formalism with the processing of exogenous events. We also study the properties of HTN acting, particularly in relation to HTN planning.

2 BACKGROUND: HTN PLANNING

In this section we provide the necessary background material on HTN planning. Some definitions are given only informally; we refer the reader to [7, 8] for the formal definitions.

An HTN *planning problem* is a tuple $\langle d, \mathcal{I}, \mathbb{D}\rangle$ comprising a *task network* d, an initial *state* $\mathcal{I}$, which is a set of ground atoms, and a *domain* $\mathbb{D} = \langle Op, Me\rangle$, where Me is a set of reduction *methods* and Op is a set of STRIPS-like operators. HTN planning involves iteratively decomposing/reducing the 'abstract tasks' occurring in d and the resulting task networks by using methods in Me, until only STRIPS-like actions remain that can be ordered and executed from $\mathcal{I}$ relative to Op.

A *task network* d is a couple $[S, \phi]$, where ϕ is a *constraint formula*, and S is a non-empty set of *labelled tasks*, i.e., constructs of the form $(n : t)$; element n is a *task label*, which is a 0-ary task-label symbol (in FOL) that is unique in d and $\mathbb{D}$, and t is a *non-primitive* or *primitive task*, which is an n-ary task symbol whose arguments are function-free terms. The *constraint formula* ϕ is a Boolean formula built from negation, disjunction, conjunction, and constraints, each of which is either: an *ordering constraint* of the form $(n \prec n')$, which requires the task (corresponding to label) n to precede task n'; a *before* (resp. an *after*) *state-constraint* of the form (l, n) (resp. (n, l)), which requires literal l to hold in the state just before (resp. after) doing n; a *between state-constraint* of the form (n, l, n'), which requires l to hold in all states between doing n and n'; or a *variable binding constraint* of the form $(o = o')$, which requires o and o' to be equal, each of which is a variable or constant. We ignore variable binding constraints as they can be specified as state-constraints, using the binary logical symbol '='.

Instead of specifying a task label, a constraint may also refer, using expression $fst[S]$ or $lst[S]$, to the action that is eventually ordered to occur first or last (respectively) among those that are yielded by the set of task labels S. While these expressions can be 'inserted' into a constraint when a task is reduced, we assume that they do not occur in methods.

A primitive task, or *action*, t, has exactly one *relevant operator* in Op, i.e., one operator associated with a primitive task t' that has the same task symbol and arity as t; any variable appearing in the operator also appears in t' and its precondition. Given a primitive task t, we denote its precondition, add-list and delete-list relative to Op as $\mathsf{pre}(t, Op)$, $\mathsf{add}(t, Op)$ and $\mathsf{del}(t, Op)$, respectively. A non-primitive task can have one or more *relevant methods* in Me. A *method* is a couple $[\![t(\mathbf{v}), d]\!]$, where $t(\mathbf{v})$ is a non-primitive task, the arguments $\mathbf{v}$ are distinct variables,[1] and d is a task network.

Given an HTN planning problem $\langle d = [S_d, \phi_d], \mathcal{I}, \langle Op, Me\rangle\rangle$, the core planning steps involve selecting a relevant method $m = [\![t_m, d_m]\!] \in Me$ for some non-primitive task $(n : t) \in S_d$ and then reducing the task to yield a 'less abstract' task network. Reducing $(n : t)$ with m involves replacing $(n : t)$ with the tasks in S_m (where $d_m = [S_m, \phi_m]$) and updating ϕ_d, e.g. to include the constraints in ϕ_m; formal definitions for method relevance and reduction are given in Section 3. The set of reductions of d is denoted $\mathsf{red}^*(d, \langle Op, Me\rangle)$.

If all non-primitive tasks in the initial and subsequent task networks have been reduced, a *completion* is obtained from the resulting 'primitive' task network. Informally, σ is a completion of a primitive task network $d = [S, \phi]$ at a state $\mathcal{I}$, denoted $\sigma \in comp(d, \mathcal{I}, \mathbb{D})$, if σ is a total ordering of a ground instance of d that satisfies ϕ; if d mentions a non-primitive task, then $comp(d, \mathcal{I}, \mathbb{D}) = \emptyset$.

Finally, the set of all HTN *solutions* is defined as $sol(d, \mathcal{I}, \mathbb{D}) = \bigcup_{n<\omega} sol_n(d, \mathcal{I}, \mathbb{D})$, where $sol_n(d, \mathcal{I}, \mathbb{D})$ is defined inductively as

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

In words, the HTN solutions for a given planning problem is the set of all completions of all primitive task networks that can be obtained via zero or more reductions of the initial task network.

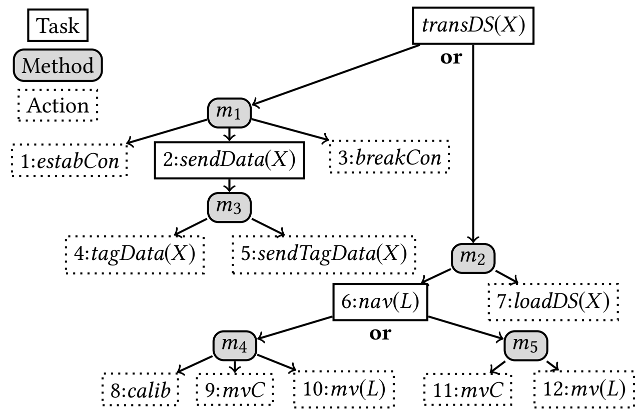
A Running Example

Let us consider the example of a rover agent exploring the surface of mars. A part of the rover's HTN domain is illustrated in Figure 1 (with braces omitted in $fst[]$ and $lst[]$ expressions). The top-level non-primitive task is to transfer, to the lander, previously gathered soil analysis data from a location X, and if possible to also deliver the soil sample for further analysis inside the lander.

The top-level task is achieved using either method m_1 or m_2, both of which require the data and sample from X to be available (i.e., for $didExp(X)$ to hold). If the rover is low on battery charge ($lowBat$), m_1 is used. This transmits the soil data but it does not deliver the soil sample, which may result in losing it if it is later discarded to make room for other samples. Method m_1 prescribes establishing radio communication with the lander, sending it the data by first including metadata, and then breaking the connection, while checking continuously that the connection is not lost between the first and last tasks (including those of m_3). If the rover is not low on battery charge, m_2 is used to achieve the top-level task; m_2 prescribes navigating to a lander L and then uploading and depositing the soil data and sample, respectively.

Navigation is performed using m_4 or m_5. Method m_4 prescribes calibrating the onboard instruments, moving the cameras to point straight (which asserts $camMoved$), and moving to the lander; while the first two actions can happen in any order, the third must happen last. The method requires that the instruments are not currently calibrated ($\neg cal$) and the battery charge is not low. Method m_5 is similar except that it is used only if the instruments are already calibrated, e.g. due to a recent calibration to achieve another task.

[1]While [8] does allow this vector to contain constants, we instead specify such binding requirements in the constraint formula.



ϕ_1	ϕ_2	ϕ_3
$(lowBat, 1)$	$(landr(L), 6)$	$(4 \prec 5)$
$(1, connEst, 3)$	$(didExp(X), 6)$	
$(didExp(X), 1)$	$(6 \prec 7)$	
$(1 \prec 2), (2 \prec 3)$		

ϕ_4	ϕ_5
$(\neg cal, fst[8, 9])$	$(cal, 11)$
$(\neg lowBat, fst[8, 9])$	$(\neg lowBat, 11)$
$(lst[8, 9] \prec 10)$	$(11 \prec 12)$

Figure 1: A partial domain for a simple rover. The tasks and methods are shown at the top, and the constraint formulas of methods are shown in the table. Each m_i is of the form $[\![t_i, d_i = [S_i, \phi_i]]\!]$. We use expressions *fst*[] and *lst*[] only for readability; it is straightforward to replace their associated constraints with those that do not contain such expressions.

Action *mv* requires $\neg lowBat$ to hold, and it consumes a significant amount of charge, i.e., it asserts *lowBat*.[2] Action *procImg* (not shown) requires *raw* and $\neg lowBat$ to hold and asserts $\neg raw$ and *lowBat*; the action processes and compresses new raw images (if any exist, i.e., *raw* holds) of the martian surface that were taken by the cameras. Doing *procImg* infrequently may result in losing older images, if they are overwritten to make space on the storage device.[3] The other actions consume a negligible amount of charge, and action *charge* (not shown) makes the battery fully charged.

3 PRELIMINARIES AND ASSUMPTIONS

In this section we formally define the notion of reduction, and we state the remaining assumptions.

First, we separate the notion of method relevance from the notion of reduction in [8]. In what follows, we use the standard notion of *substitution* [15], and of *applying* a substitution θ to an expression E, which we denote by $E\theta$.

Definition 3.1 (Relevant Method). Let $\mathbb{D} = \langle Op, Me \rangle$ be a domain, t a non-primitive task, and $[\![t', d]\!] \in Me$ a method. If $t = t'\theta$ for some substitution θ, then $d\theta$ is a ***relevant*** method-body for t relative to $\mathbb{D}$.[4] The set of all such method-bodies is denoted by $\mathsf{rel}(t, \mathbb{D})$.

In the definition of reduction below, and in the rest of the paper, we denote by $\mathsf{lab}(S)$ the set of all task labels appearing in a given set of labelled tasks S.

Definition 3.2 (Reduction (adapted from [8])). Let $d = [\{(n : t)\} \cup S, \phi]$, with $(n : t) \notin S$, be a task network and t a non-primitive task, and let $d' = [S', \phi'] \in \mathsf{rel}(t, \mathbb{D})$. The ***reduction*** of n in d with d', denoted $\mathsf{red}(d, n, d')$, is the task network $[S \cup S', \phi' \wedge \psi]$, where ψ is obtained from ϕ with the following modifications:

- replace $(n \prec n_j)$ with $(lst[\mathsf{lab}(S')] \prec n_j)$, as n_j must come after every task in n's decomposition;
- replace $(n_j \prec n)$ with $(n_j \prec fst[\mathsf{lab}(S')])$;
- replace (l, n) with $(l, fst[\mathsf{lab}(S')])$, as l must be true immediately before the first task in n's decomposition;
- replace (n, l) with $(lst[\mathsf{lab}(S')], l)$, as l must be true immediately after the last task in n's decomposition;
- replace (n, l, n_j) with $(lst[\mathsf{lab}(S')], l, n_j)$;
- replace (n_j, l, n) with $(n_j, l, fst[\mathsf{lab}(S')])$; and
- everywhere that n appears in ϕ in a $fst[]$ or a $lst[]$ expression, replace it with $\mathsf{lab}(S')$.

For example, consider task network $d = [S, \phi]$, where $S = \{(A : transDS(loc1)), (B : charge)\}$ and $\phi = (A \prec B)$. Observe that method m_2 in Figure 1 is $[\![transDS(X), d_2 = [S_2, \phi_2]]\!]$, where $S_2 = \{(6 : nav(L)), (7 : loadDS(X))\}$ and $\phi_2 = (landr(L), 6) \wedge (didExp(X), 6) \wedge (6 \prec 7)$. Then, $\mathsf{red}(d, A, d_2)$ is task network $[S', \phi']$ where $S' = \{(6 : nav(L)), (7 : loadDS(loc1)), (B : charge)\}$ and ϕ' is the conjunction of ϕ_2 and ϕ updated to account for the reduction, i.e., $\phi' = (landr(L), 6) \wedge (didExp(loc1), 6) \wedge (6 \prec 7) \wedge (lst[6, 7] \prec B)$.

In the rest of the paper, we ignore the *charge* task, and when we need to refer to a labelled task we simply use its task label if the corresponding task is obvious; e.g. we would represent S' above as $\{6, (7 : loadDS(loc1)), B\}$.

The remaining assumptions that we make are the following. First, without loss of generality [4], we assume that HTN domains are *conjunctive*, i.e., they do not mention constraint formulas that specify a disjunction of elements. Thus, we sometimes treat a constraint formula as a set (of possibly negated constraints).

Definition 3.3 (Conjunctive HTNs [4]). A task network $[S, \phi]$ is ***conjunctive*** if its constraint formula ϕ is a conjunction of possibly negated constraints. A ***domain*** $\langle Op, Me \rangle$ is ***conjunctive*** if the task network d in every method $[\![t, d]\!] \in Me$ is conjunctive.

Second, to distinguish between reductions that are being pursued at different levels of abstraction, we assume that a reduction produces at least two tasks, i.e., any method $[\![t, [S, \phi]]\!] \in Me$ is such that $|S| > 1$. This can be achieved using 'no-op' actions, denoted *nop*, if necessary, which have 'empty' preconditions and effects.

Third, for any method $[\![t, [S, \phi]]\!] \in Me$, there exists a (possibly 'no-op') task $(n : t) \in S$ such that $(n' \prec n) \in \phi$ for any $n' \in \mathsf{lab}(S) \setminus \{n\}$, and $(n, l) \notin \phi$ for any l. This will ensure that all the after state-constraints in ϕ are evaluated by our semantics.

[2]For simplicity, we assume 'low charge' is less than or equal to 50% of the maximum charge, and an action requiring a 'significant' amount of charge consumes 50%. We also consider it unsafe for the charge to reach 0%.

[3]We assume that delivering a soil sample to the lander and processing images before they are overwritten have equal importance.

[4]All variables and task labels in $d\theta$ must be renamed with variables and task labels that do not appear anywhere else.

Finally, we assume that the user does not specify inconsistent ordering constraints in a method's constraint formula, e.g. the constraints $(1 \prec 2), (2 \prec 3)$, and $(3 \prec 1)$. Formally, let ϕ^* denote the transitive closure of a constraint formula ϕ, i.e., the one that is obtained from ϕ by adding the constraint $(n_1 \prec n_{i+1})$ whenever $(n_1 \prec n_2), (n_2 \prec n_3), \ldots, (n_i \prec n_{i+1}) \in \phi$ holds for some $i > 1$. Then, for any method $\llbracket t, [S, \phi] \rrbracket \in Me$, there does not exist a pair $(n \prec n'), (n' \prec n) \in \phi^*$ nor $(n \prec n'), \neg(n \prec n') \in \phi^*$.

4 A FORMALISM FOR HTN ACTING

We now develop a formalism for HTN acting by defining, in particular, three notions of execution: via *reduction*, *action*, and *replacement*. The first notion is based on task reduction; the second notion defines what it means to execute an action in the HTN setting, in particular, the gathering and evaluating of constraints that are relevant to the action; and the third notion represents failure handling, i.e., the replacement of 'blocked' tasks by alternative ones.

We only allow a task occurring in a task network to be executed via action or reduction if it is a *primary* task in the network, i.e., there are no other tasks that *must* precede it. Formally, given a task network $d = [S, \phi]$, we first define the following sets of tasks:

$$\begin{aligned} S_1 &= \{(n : t) \in S \mid (x \prec x') \in \phi, n \text{ occurs in } x'\},\\ S_2 &= \{(n : t) \in S \mid t \text{ is an action and either } \neg(n \prec x) \in \phi \text{ or}\\ &\qquad \neg(lst[\{n\}] \prec x) \in \phi\}. \end{aligned}$$

That is, S_1 and S_2 contain the tasks that *cannot* be primary ones; the above action n occurring in a negated ordering constraint cannot be a primary task because one or more tasks (represented by x above) must precede n.[5] Then, we define the set of ***primary*** tasks of task network d as $\mathsf{primary}(d) = S \setminus (S_1 \cup S_2)$. For example, given task network d_1 in method m_1 in Figure 1, $\mathsf{primary}(d_1) = \{1\}$, and given task network d_4 in method m_4, $\mathsf{primary}(d_4) = \{8, 9\}$.

We can now define our first notion, an *execution via reduction* of a task network, as the reduction of an arbitrary primary non-primitive task via a relevant method. To enable trying alternative reductions for the task if the one that was selected fails or is not applicable, we maintain the set of all relevant methods for the task, and update the set as alternative methods are tried. We use the term *reduction couple* to refer to a couple comprising two sets: *(i)* the set representing the reductions being pursued for a task (and its subtasks), and *(ii)* the set of current alternative method-bodies for the task. We use R to denote the set of reduction couples corresponding to the tasks reduced so far, where each couple is of the form $\langle S, D \rangle$, with S being a set of labelled tasks, and D a set of task networks. While the initial value of R and how it can 'evolve' will be made concrete via formal definitions, we shall for now illustrate these with an example.

Let us consider the task network $[S, \phi = true]$, where the set $S = \{(A : transDS(loc1)), (B : procImg)\}$; the initial state $\mathcal{I} = \{raw, cal, didExp(loc1), landr(lan1)\}$; the 'initial' set of reduction couples $R = \{\langle S, \emptyset \rangle\}$; and the domain $\mathbb{D}$ is as depicted in Figure 1. An execution via reduction of the task network from $\mathcal{I}$ relative to R and $\mathbb{D}$ is the tuple $\langle [S', \phi'], \mathcal{I}, R', \mathbb{D} \rangle$, where $S' = \{6, 7, B\}$, formula ϕ' is ϕ_2 in Figure 1 with variable X substituted with $loc1$, and the resulting set of reduction couples $R' = \{\langle S', \emptyset \rangle, \langle \{6, 7\}, \{d_1\} \rangle\}$, where d_1 is the alternative method-body for A. Moreover, an execution via reduction of $[S', \phi']$ is the tuple $\langle [S'', \phi''], \mathcal{I}, R'', \mathbb{D} \rangle$, where $S'' = S''' \cup \{7, B\}$, set $S''' = \{11, 12\}$, formula ϕ'' is the conjunction of ϕ_5 and ϕ' updated to account for the reduction, and set $R'' = \{\langle S'', \emptyset \rangle, \langle S''' \cup \{7\}, \{d_1\} \rangle, \langle S''', \{d_4\} \rangle\}$.

We call a 4-tuple of the form $\langle d, \mathcal{I}, R, \mathbb{D} \rangle$, as in the example above, a ***configuration***. (For brevity, we omit the fifth element θ, representing the substitutions applied so far to variables appearing in d.) Formally, we define an execution via reduction as follows.

Definition 4.1 (Execution via Reduction). Let $\mathbb{D}$ be a domain; $\mathcal{I}$ a state; d a task network with a non-primitive task $(n : t) \in \mathsf{primary}(d)$; R a set of reduction couples; $d_n = [S_n, \phi_n] \in D$ a method-body, with $D = \mathsf{rel}(t, \mathbb{D})$; and couple $r = \langle S_n, D \setminus \{d_n\} \rangle$. An ***execution*** via ***reduction*** of d from $\mathcal{I}$ relative to R and $\mathbb{D}$ is the configuration $\langle \mathsf{red}(d, n, d_n), \mathcal{I}, R' \cup \{r\}, \mathbb{D} \rangle$, where R' is R with any occurrence of $(n : t)$ replaced by the elements in set S_n.

We now define the second kind of execution: performing an action. In order to execute a (primary) action, it must be *applicable*, i.e., its precondition and any constraints that are *relevant* to the action must hold in the current state. Such constraints could have been (directly) specified on the action, 'inherited' from one or more of the action's 'ancestors', or 'propagated' from an earlier action. We first define the notion of a relevant constraint; we ignore negated between state-constraints for brevity.[6]

Definition 4.2 (Relevant Constraint). Let $d = [S, \phi]$ be a task network with an action $(n : t) \in S$, and $c \in \phi$ a between state-constraint or a possibly negated before or after state-constraint. Let c_2 be the non-negated constraint corresponding to c. Then, c is ***relevant*** for executing n relative to d if for some literal l:

- $c_2 \in \{(l, n), (l, fst[\{n, \ldots\}])\}$; or

  for some x' and $n' \notin \mathsf{lab}(S)$,
- $c_2 \in \{(n', l), (lst[\emptyset], l), (n', l, x'), (lst[\emptyset], l, x')\}$.

The set of relevant constraints for executing n relative to d is denoted by $\mathsf{bef}(n, d)$. For example, if d is the resulting task network after the two reductions in our running example, the relevant constraints for $(11 : mvC)$ in Figure 1 is the set: $\{(landr(L), fst[11, 12]), (didExp(loc1), fst[11, 12]), (\neg lowBat, 11), (cal, 11)\}$, where the first two constraints are 'inherited' from $(6 : nav(L))$. In the above definition, n' and $lst[\emptyset]$ represent an action that was already executed, whose associated after or between state-constraints have been 'propagated' to n.

We next define what it means to 'extract' the literals from a given set of state constraints. Let us denote the subset of negated constraints as $\mathsf{bef}^-(n, d) = \{c \in \mathsf{bef}(n, d) \mid c \text{ is a negated constraint}\}$, and the subset of positive ones as $\mathsf{bef}^+(n, d) = \mathsf{bef}(n, d) \setminus \mathsf{bef}^-(n, d)$. Then, the set of *extracted literals* is denoted $\mathsf{bef}_l(n, d) = \{l \mid \text{literal } l \text{ occurs in } c, c \in \mathsf{bef}^+(n, d)\} \cup \{\neg l \mid \text{literal } l \text{ occurs in } c, c \in \mathsf{bef}^-(n, d)\}$. We can now define what it means for an action to be applicable.

Definition 4.3 (Applicability). Let $\mathbb{D} = \langle Op, Me \rangle$ be a domain, $\mathcal{I}$ a state, and $d = [S, \phi]$ a task network with an action $(n : t) \in S$

[5] This is provided none of the actions associated with x have already been executed. As we show later, in our semantics, such an execution will result in the (then 'realised') constraint being removed.

[6] To account for a constraint $\neg(n_1, l, n_2)$, we check in every state between n_1 and n_2 whether $\neg l$ holds. If so, we remove the constraint from the formula. If $\neg(n_1, l, n_2)$ exists when the first action of n_2 is executed, $\neg l$ is then relevant for it.

such that $n \in \mathsf{primary}(d)$. Let $\Phi(n, d, Op)$ denote the precondition and extracted literals, i.e., the formula $\mathsf{pre}(t, Op) \wedge \bigwedge l \in \mathsf{bef}_l(n, d)$. Then, n is <u>*applicable*</u> in $\mathcal{I}$ relative to d and Op if $\mathcal{I} \models \Phi(n, d, Op)$.

Executing an applicable action results in changes to both the current state and the current task network: the action is removed from the network's set of tasks, and the action's 'realised' constraints, e.g. the relevant ones that do not need to be re-evaluated before executing other actions, are removed from the network's constraint formula. The constraints that do need to be re-evaluated are the between state-constraints that require literals to hold from the end of an action that was executed earlier, up to an action that is yet to be executed. Formally, given a task network $d = [S, \phi]$ and an action $(n : t) \in S$, we denote by C_1 the *realised ordering constraints* upon executing n (relative to d), i.e., the set

$$\begin{array}{lll}\{(x \prec x') \in \phi & | & \text{for some } x' \text{ and } x \in \{n, lst[\{n\}]\}\} \cup \\ \{\neg(x' \prec x) \in \phi & | & \text{for some } x' \text{ and } x \in \{n, fst[\{n, \ldots\}]\}\},\end{array}$$

where x' represents an action(s) that is yet to be executed. Notice that a negated ordering constraint is realised only if one or more (or all) of the actions corresponding to x' are executed after the first (or only) one corresponding to x. Next, we denote by C_2 the *realised state constraints* upon executing n, i.e., the set obtained from $\mathsf{bef}(n, d)$ by removing any between state-constraint (x, l, x') when $x' \neq n$ and $x' \neq fst[\{n, \ldots\}]$. Then, we can define the set of <u>*realised*</u> constraints upon executing n relative to d as $\mathsf{fin}(n, d) = C_1 \cup C_2$, and the result of executing an action as follows.

Definition 4.4 (Action Result). Let Op be a set of operators, $\mathcal{I}$ a state, d a task network, R a set of reduction couples, $(n : t) \in \mathsf{primary}(d)$ an action, and θ a substitution. The <u>*result*</u> of executing n from $\mathcal{I}$ relative to d, θ, R and Op, denoted $\mathsf{res}(n, \mathcal{I}, d, \theta, R, Op)$, is the tuple $\langle [S', \phi'], \mathcal{I}', R\rangle\theta$, where

- $S' = S \setminus \{(n : t)\}$, where $d = [S, \phi]$;
- $\mathcal{I}' = (\mathcal{I} \setminus \mathsf{del}(t\theta, Op)) \cup \mathsf{add}(t\theta, Op)$; and
- ϕ' is obtained from $\phi \setminus \mathsf{fin}(n, d)$ by removing all occurrences of n within $lst[]$ expressions.[7]

Notice that the only possible update to R is a substitution of one or more variables (we do not remove executed actions from reduction couples). Finally, we define an execution via action of a task network as the execution of (a ground instance of) an applicable primary action in it.

Definition 4.5 (Execution via Action). Let $\mathbb{D} = \langle Op, Me\rangle$ be a domain, $\mathcal{I}$ a state, R a set of reduction couples, and $d = [S, \phi]$ a task network such that $\mathcal{I} \models \psi$ for some θ and action $(n : t) \in \mathsf{primary}(d)$, where $\psi = \Phi(n, d, Op)\theta$ is a ground formula. An <u>*execution*</u> via <u>*action*</u> of d from $\mathcal{I}$ relative to R and $\mathbb{D}$ is the configuration $\langle d', \mathcal{I}', R', \mathbb{D}\rangle$, where $\langle d', \mathcal{I}', R'\rangle = \mathsf{res}(n, \mathcal{I}, d, \theta, R, Op)$.

Continuing with our running example, let $\langle [S, \phi], \mathcal{I}, R, \mathbb{D}\rangle$, with $S = \{11, 12, 7, B\}$, be the configuration resulting from the two reductions from before. Then, an execution via action of d from $\mathcal{I}$ relative to R and $\mathbb{D}$ is the configuration $\langle [S', \phi'], \mathcal{I}', R', \mathbb{D}\rangle$, where $\mathcal{I}' = \mathcal{I} \cup \{camMoved\}$; set $S' = \{(12 : mv(lan1)), 7, B\}$; formula ϕ' is obtained from ϕ by removing all constraints except for $(lst[11, 12] \prec 7)$, which is updated to $(lst[12] \prec 7)$; and R' is obtained from R by applying substitution $\{L/lan1\}$.

Observe that the applicability of a method (relative to the current state) is not checked at the point that it is chosen to reduce a task, but *immediately* before executing (for the first time) an associated primary action—which may be after performing further reductions and unordered actions. On the other hand, BDI agent programming languages such as AgentSpeak and CAN check the applicability of a relevant recipe at *some point* before (not necessarily just before) executing an associated primary action. Thus, in cases where the environment changes between checking the recipe's applicability and executing an associated primary action (for the first time), and makes the recipe no longer applicable, the action will still be executed (provided, of course, the action itself is applicable). Such behaviour is not permitted by our semantics.

We now define the final notion of execution: *execution via replacement*, i.e., replacing the reductions being pursued for a task if they have become *blocked*. Intuitively, this happens when none of the primary actions in the pursued reductions are applicable, and none of the primary non-primitive tasks have a relevant method-body.

Formally, let $\mathbb{D} = \langle Op, Me\rangle$ be a domain, $\mathcal{I}$ a state, d a task network, and $\langle S, D\rangle$ a reduction couple with $S \cap \mathsf{primary}(d) \neq \emptyset$. Then, set S is <u>*blocked*</u> in d from $\mathcal{I}$ relative to $\mathbb{D}$, denoted $\mathsf{blocked}(S, d, \mathcal{I}, \mathbb{D})$, if for all $(n : t) \in S \cap \mathsf{primary}(d)$, either t is an action and $\mathcal{I} \not\models \Phi(n, d, Op)$, or t is a non-primitive task and $\mathsf{rel}(t, \mathbb{D}) = \emptyset$. Recall that S represents the reductions that are being pursued for a particular task (and its subtasks).

When such pursued reductions are blocked, they are replaced by an alternative relevant method-body for the task. In the definition below, we use the $fst[]$ and $lst[]$ constructs (if any) 'inserted' into the constraint formula by the first reduction of the task (Definition 3.2). Recall that these constructs represent the 'inheritance' of the task's associated constraints by its descendant tasks.

Definition 4.6 (Replacement). Let $d = [S_d, \phi_d]$ be a task network, $\langle S, D\rangle$ a reduction couple, and $d_{new} = [S_{new}, \phi_{new}] \in D$. The <u>*replacement*</u> of (the elements of) S in S_d with S_{new} relative to d_{new} and d, denoted $\mathsf{rep}(S, d_{new}, d)$, is the task network

$$[(S_d \setminus S') \cup S_{new}, \psi \wedge \phi_{new}],$$

where $S' = S \cap S_d$, and ψ is obtained from ϕ_d by *(i)* replacing any occurrence of (all) the task labels in $\mathsf{lab}(S')$—within a $fst[]$ or a $lst[]$ expression—with the labels in $\mathsf{lab}(S_{new})$, and then *(ii)* removing any element mentioning a task label in $\mathsf{lab}(S)$.

After a replacement, we need to *update* the set of reduction couples accordingly, by doing the same replacement in all relevant reduction couples. In the definition below, the set S' and task network d_{new} are as above.

Definition 4.7 (Update). Let R be a set of reduction couples with $\langle S, D\rangle \in R$, let $S' \subseteq S$, and $d_{new} \in D$. The <u>*update*</u> of S' in S with S_{new} relative to d_{new} and R, denoted $\mathsf{upd}(S', S, d_{new}, R)$, is the set obtained from $R' = (R \setminus \{\langle S, D\rangle\}) \cup \{\langle S, D \setminus \{d_{new}\}\rangle\}$ by replacing any couple $\langle S'' \supseteq S, D''\rangle$ with $\langle (S'' \setminus S') \cup S_{new}, D''\rangle$, and then removing any couple that still mentions an element in S'.

Finally, we combine the two definitions above to define the configuration that results from an execution via replacement. While we

[7] We also remove from ϕ' any (remaining) constraint of the form (x, l, x') such that n occurs in x', i.e., a between state-constraint that holds trivially.

provide a general definition, for replacing *any* task's blocked (pursued) reductions, one might instead want to, as in depth-first search, first replace a *least abstract* task's blocked reductions. That is, one might want to first consider the *smallest replaceable* reduction couples. Formally, given a set of reduction couples R, a couple $\langle S, D\rangle \in R$ is a <u>*smallest replaceable*</u> one in R, denoted $\langle S, D\rangle \in \mathsf{smallest}(R)$, if $D \neq \emptyset$ and for each couple $\langle S', D'\rangle \in R$, either *(a)* $S' \supseteq S$; *(b)* $S' \subset S$ and $D' = \emptyset$; or *(c)* $S \cap S' = \emptyset$.

Definition 4.8 (Execution via Replacement). Let $\mathbb{D}$ be a domain, $\mathcal{I}$ a state, $d = [S_d, \phi_d]$ a task network, and R a set of reduction couples with an $r = \langle S, \{d_{new}, \ldots\}\rangle \in R$ such that $\mathsf{blocked}(S, d, \mathcal{I}, \mathbb{D})$ holds. An <u>*execution*</u> via <u>*replacement*</u> of d from $\mathcal{I}$ relative to R and $\mathbb{D}$ is the configuration

$$\langle \mathsf{rep}(S, d_{new}, d), \mathcal{I}, \mathsf{upd}(S \cap S_d, S, d_{new}, R), \mathbb{D}\rangle;$$

the replacement is <u>*complete*</u> if $S \subseteq S_d$ and <u>*partial*</u> otherwise, and a <u>*jump*</u> if $r \notin \mathsf{smallest}(R)$.

A complete-replacement represents the BDI-style searching of an achievement-goal's (i.e., a task's) set of relevant recipes in order to find one that is applicable, and a partial-replacement represents BDI-style recovery from the failure to execute (or successfully execute) an action, e.g. due to an environmental change. We illustrate these two notions of replacement with the following examples.

Continuing with our running example, let $\langle [S, \phi], \mathcal{I}, R, \mathbb{D}\rangle$ be the configuration resulting from the two reductions from before. Suppose however that the rover's instruments were not calibrated, i.e., $\mathcal{I} \not\models cal$. Then, action $(11 : mvC)$ is not applicable, and an execution via complete-replacement is performed on tasks in $[S, \phi]$ to obtain configuration $\langle [S', \phi'], \mathcal{I}, R', \mathbb{D}\rangle$, where $S' = S'' \cup \{7, B\}$; set $S'' = \{8, 9, 10\}$; formula ϕ' is the conjunction of ϕ_4, and ϕ updated by, e.g. removing the constraints that were copied from ϕ_5 and replacing constraint $(landr(L), fst[11, 12])$ with $(landr(L), fst[8, 9, 10])$; and the set of couples $R' = \{\langle S', \emptyset\rangle, \langle S'' \cup \{7\}, \{d_1\}\rangle, \langle S'', \emptyset\rangle\}$.

Suppose we now perform two executions via action to obtain configuration $\langle [S''', \phi''], \mathcal{I}', R'', \mathbb{D}\rangle$, with $S''' = \{(10 : mv(lan1)), 7, B\}$ and formula ϕ'' (resp. set R'') being ϕ' (resp. R') updated to account for the executions. Finally, suppose that the battery level drops due to the execution of top-level image processing action $(B : procImg)$, which makes $mv(lan1)$ no longer applicable. (We will show later how *procImg* could instead be absent in the initial task network and arrive 'dynamically' from the environment.) Then, an execution via partial-replacement will be performed on tasks in $[S''' \setminus \{(B : procImg)\}, \phi'']$ to obtain configuration $\langle [\{1, 2, 3\}, \phi'''], \mathcal{I}'', R''', \mathbb{D}\rangle$, where ϕ''' (resp. $\mathcal{I}''$) is the updated ϕ'' (resp. $\mathcal{I}'$), and the set $R''' = \{\langle \{8, 9, 1, 2, 3, B\}, \emptyset\rangle, \langle \{8, 9, 1, 2, 3\}, \emptyset\rangle\}$.

5 PROPERTIES OF THE FORMALISM

In this section, we discuss the properties of our formalism, and in particular how it relates to HTN planning.

The properties are based on the definition of an *execution trace*, which formalises the consecutive execution of a configuration—via reduction, replacement, or action—as in our running example. In what follows, we use $\tau \in \mathsf{exec}(d, \mathcal{I}, R, \mathbb{D})$ to denote that a configuration τ is an execution via reduction, action, or replacement of a task network d from a state $\mathcal{I}$ relative to a set of reduction couples R and a domain $\mathbb{D}$.

Definition 5.1 (Execution Trace). Let $d = [S_d, \phi_d]$ be a task network, $\mathcal{I}$ a state, and $\mathbb{D}$ a domain. An <u>*execution trace*</u> $\mathcal{T}$ of d from $\mathcal{I}$ relative to $\mathbb{D}$ is any sequence of configurations $\tau_1 \cdot \ldots \cdot \tau_k$, with each $\tau_i = \langle d_i, \mathcal{I}_i, R_i, \mathbb{D}\rangle$, such that $d_1 = d$; $\mathcal{I}_1 = \mathcal{I}$; $R_1 = \{\langle S_d, \emptyset\rangle\}$; and $\tau_{i+1} \in \mathsf{exec}(d_i, \mathcal{I}_i, R_i, \mathbb{D})$ for all $i \in [1, k-1]$.

We also need some auxiliary definitions related to execution traces. Consider configuration τ_k above. First, if $S_k = \emptyset$ (where $d_k = [S_k, \phi_k]$), then the trace is <u>*successful*</u>. Second, if for all couples $\langle S, D\rangle \in R_k$ we have that $S \cap \mathsf{primary}(d_k) \neq \emptyset$ entails both $\mathsf{blocked}(S, d_k, \mathcal{I}_k, \mathbb{D})$ and $D = \emptyset$, then the trace is <u>*blocked*</u>. The following theorem states that if a trace is successful or blocked as we have 'syntactically' defined, then there is no way to 'extend' the trace further, and vice versa.

Proposition 5.2. *Let $\mathcal{T}$ be an execution trace of a task network d from a state $\mathcal{I}$ relative to a domain $\mathbb{D}$. There exists an execution trace $\mathcal{T} \cdot \tau_1 \cdot \ldots \cdot \tau_k$, with $k > 0$, of d (from $\mathcal{I}$ relative to $\mathbb{D}$) if and only if $\mathcal{T}$ is neither successful nor blocked. The inverse also holds.*

Proof. If there exists a trace $\mathcal{T} \cdot \tau_1 \cdot \ldots \cdot \tau_k$ with $k > 0$ then $\mathcal{T}$ cannot be successful as its final task network $d_k = [S_k, \phi_k]$ would then not mention any tasks, and thus we cannot 'extend' it to τ_1. The fact that $\mathcal{T}$ cannot be blocked follows from the fact that an execution via replacement, action, or reduction of d_k is possible. Conversely, if $\mathcal{T}$ is neither successful nor blocked, then the only reason it would not be possible to 'extend' it is if $S_k \neq \emptyset$ but $\mathsf{primary}(d_k) = \emptyset$. However, this is only possible if a method-body exists where its constraint formula contains inconsistent (possibly negated) ordering constraints. Such method-bodies are not allowed due to our assumption in Section 3. The inverse of the theorem is proved similarly. □

The next three properties rely on traces that are free from certain kinds of execution. A trace $\mathcal{T} = \tau_1 \cdot \ldots \cdot \tau_k$ is <u>*complete-replacement free*</u> if there does not exist an index $i \in [1, k-1]$ such that τ_{i+1} is an execution via complete-replacement of d_i from $\mathcal{I}_i$ relative to R_i and $\mathbb{D}$. We define *partial-replacement free* and *jump free* traces similarly.

Given any execution trace, the next theorem states that there is an equivalent one—in terms of actions performed—that is complete-replacement free. Intuitively, this is because, either with some 'lookahead' mechanism or 'luck', a complete-replacement can be avoided by choosing a different (or 'correct') relevant method-body for a task. We define the *actions performed* by a trace $\mathcal{T} = \tau_1 \cdot \ldots \cdot \tau_k$ (or the pursued 'solution'), denoted $\mathsf{act}(\mathcal{T})$, as follows. Given an index $i \in [1, k-1]$, we first define $\mathsf{act}(i) = t$ if $S_i \setminus S_{i+1} = \{(n : t)\}$ and τ_{i+1} is an execution via action of d_i from $\mathcal{I}_i$ relative to R_i and $\mathbb{D}$; otherwise, we define $\mathsf{act}(i) = \epsilon$. Then, $\mathsf{act}(\mathcal{T})$ is $\mathsf{act}(1) \cdot \ldots \cdot \mathsf{act}(k-1)$ with substitution θ of configuration τ_k applied to the sequence.

Theorem 5.3. *Let $\mathcal{T}$ be an execution trace of a task network d from a state $\mathcal{I}$ relative to a domain $\mathbb{D}$. There exists a complete-replacement free execution trace $\mathcal{T}'$ of d from $\mathcal{I}$ relative to $\mathbb{D}$ such that $\mathsf{act}(\mathcal{T}) = \mathsf{act}(\mathcal{T}')$ and $|\mathcal{T}'| \leq |\mathcal{T}|$.*

Proof Sketch. Let $\mathcal{T} = \tau_1 \cdot \ldots \cdot \tau_k$. The main steps are as follows. Consider the first execution via complete-replacement, i.e., the smallest $0 < m < k$ such that τ_{m+1} (with each $\tau_j = \langle d_j, \mathcal{I}_j, R_j, \mathbb{D}\rangle$) is an execution via complete-replacement of d_m from $\mathcal{I}_m$ relative to R_m and $\mathbb{D}$. If there is no such τ_{m+1}, the theorem holds; otherwise, $d_{m+1} = \mathsf{rep}(\underline{S}, \underline{d}', d_m)$ for some $\langle \underline{S}, D\rangle \in R_m$ and $\underline{d}' \in D$ (Def. 4.6). We show how this replacement can be 'removed' from the trace.

Consider prefix $\tau_1 \cdot \ldots \cdot \tau_i$ with $i < m$, where τ_i is where the (first) 'incorrect' reduction, which led to the replacement, was performed. Let $d_{i+1} = \mathsf{red}(d_i, \underline{n}, \underline{d})$ be the reduction, for some $(\underline{n} : t) \in S_i$ and $\underline{d} \in \mathsf{rel}(t, \mathbb{D})$, with $d_i = [S_i, \phi_i]$. Further, let $\tau' = \langle d_{\tau'}, \mathcal{I}_{\tau'}, R_{\tau'}, \mathbb{D}\rangle$ represent the alternative reduction corresponding to replacement $\underline{d}'$ above, i.e., the execution via reduction of d_i from $\mathcal{I}_i$ relative to R_i and $\mathbb{D}$ such that $d_{\tau'} = \mathsf{red}(d_i, \underline{n}, \underline{d}')$. We now show that all 'relevant' executions from τ_{i+1} up to τ_m (which cannot be complete-replacements) are also possible from τ', and thereby that τ_{m+1} can be reached from τ' without the complete-replacement.

Consider one such execution. If $i + 1 < m$, then τ_{i+2} is an execution via reduction, partial-replacement or action of d_{i+1} from $\mathcal{I}_{i+1}$ relative to R_{i+1} and $\mathbb{D}$. Let $S_{bef} \subset S_{i+1}$ be the task that was executed or reduced, or the tasks that were replaced. If $S_{bef} \subset S_{\tau'}$ (where $d_{\tau'} = [S_{\tau'}, \phi_{\tau'}]$) then the execution is 'relevant' to $d_{\tau'}$, i.e., the execution does not involve an 'ancestor' of $\underline{S}$. We then show that there also exists a configuration τ'' that is an execution via reduction, partial-replacement or action of $d_{\tau'}$ (from $\mathcal{I}_{\tau'}$ relative to $R_{\tau'}$ and $\mathbb{D}$) that involves the same tasks as in S_{bef}. In particular, we show that the difference (if any) between formula $\phi_{\tau'}$ and ϕ_{i+1}, due to the different reductions performed on d_i, does not affect the above execution. □

An equivalent complete-replacement free trace may, however, unavoidably specify one or more replacements that are jumps—where the smallest replaceable reduction couples were skipped. To see why this holds, consider once again our running example, but suppose that the constraints associated with $\neg lowBat$ do not exist in ϕ_4 and ϕ_5 in Figure 1.[8] Suppose also that after the first reduction (of task A), task 6 is reduced using method m_4 instead of m_5, which means that the complete-replacement in the previous example will not occur. The resulting set of reduction couples will then contain the couple $\langle S, \{d_5\}\rangle$, with $S = \{8, 9, 10\}$, instead of the couple $\langle S, \emptyset\rangle$ in the previous example (after the complete-replacement was performed). Thus, after the two executions via action of tasks 8 and 9 as before, the subsequent partial-replacement must 'skip' couple $\langle S, \{d_5\}\rangle$, which is the smallest replaceable one, and 'jump' to couple $\langle S \cup \{7\}, \{d_1\}\rangle$ in order to avoid performing $(11 : mvC)$. Intuitively, the jump is needed to 'mimic' the actions yielded by the trace depicted by the previous example, which considered d_5 but then removed it (via the complete-replacement) because it was not applicable. This observation is stated formally below.

Proposition 5.4. *There exists a domain $\mathbb{D}$, state $\mathcal{I}$, task network d, and an execution trace $\mathcal{T}$ of d from $\mathcal{I}$ relative to $\mathbb{D}$ such that any complete-replacement free execution trace $\mathcal{T}'$ of d from $\mathcal{I}$ relative to $\mathbb{D}$ is not jump free when* $\mathsf{act}(\mathcal{T}) = \mathsf{act}(\mathcal{T}')$.

Proof. This follows from the example above. □

The next result makes the link concrete between our HTN acting formalism and HTN planning. It states that the solution yielded by any execution trace that is successful and free from partial-replacements can also be yielded via HTN planning. Conversely, given any HTN planning solution, there exists such an execution trace that yields it. The trace must be free from partial-replacements because such behaviour is specific to BDI-style recovery from runtime failure.

Theorem 5.5. *Let $\mathbb{D}$ be a domain, $\mathcal{I}$ a state, and d a task network. Then, $\sigma \in \mathsf{sol}(d, \mathcal{I}, \mathbb{D})$ if and only if there exists a partial-replacement free and successful execution trace $\mathcal{T}$ of d from $\mathcal{I}$ relative to $\mathbb{D}$ such that $\sigma = \mathsf{act}(\mathcal{T})$.*

Proof Sketch. We prove this by induction on the length of the prefixes of σ. We sketch only one direction of the proof: where $\sigma \in \mathsf{sol}(d, \mathcal{I}, \mathbb{D})$. Observe that by the definition of $sol(d, \mathcal{I}, \mathbb{D})$ in Section 2, there exists a sequence $\mathbf{d} = d_1 \cdot \ldots \cdot d_m$ with $d = d_1$, and in particular with $\sigma \in \mathsf{comp}(d_m, \mathcal{I}, \mathbb{D})$. Informally, given a task label n, let function $f(n)$ denote the index in $\mathbf{d}$ where n first appears.

For the base case, we consider the labelled action $(\underline{n} : \underline{t})$ corresponding to the first action in σ. Let $\tau_1 \cdot \ldots \cdot \tau_k$, with $k \leq m$ and each $\tau_i = \langle d_i = [S_i, \phi_i], \mathcal{I}, R_i, \mathbb{D}\rangle$, be a trace of d from $\mathcal{I}$ relative to $\mathbb{D}$ such that *(i)* $k = f(\underline{n})$, and *(ii)* for each $i \in [1, k-1]$, $d_{i+1} = \mathsf{red}(d_i, n_i, \hat{d}_i)$ for some $(n_i, t_i) \in S_i$ and $\hat{d}_i \in \mathsf{rel}(t_i, \mathbb{D})$. Since $(\underline{n} : \underline{t})$ is the first action in σ (which is applicable in $\mathcal{I}$), the precondition of $\underline{t}$ holds in $\mathcal{I}$, and so does any possibly negated constraint in ϕ_m of the form $(l, \underline{n})$ or $(l, fst[\underline{n}, \ldots])$. Therefore, $\mathcal{I} \models \Phi(\underline{n}, d_k, Op)$ also holds (Def. 4.3): any such constraints will also occur in ϕ_k and no more before state-constraints can occur in ϕ_k that are associated with $\underline{n}$. Then, we can take trace $\mathcal{T}^1 = \tau_1 \cdot \ldots \cdot \tau_k \cdot \langle [S', \phi'], \mathcal{I}', \mathcal{R}', \mathbb{D}\rangle$, where the last configuration is an execution via action of d_k from $\mathcal{I}$ (relative to R_k and $\mathbb{D}$) such that $\underline{n} \in \mathsf{lab}(S_k)$ but $\underline{n} \notin \mathsf{lab}(S')$. From this it follows that the theorem holds in the base case.

For the induction hypothesis, we assume that the theorem holds for any prefix of σ of length up to $\ell < |\sigma|$. We then show that the theorem also holds for the prefix of σ of length $\ell + 1$, using, from the induction hypothesis, the fact that there is a trace $\mathcal{T}^\ell$ of d from $\mathcal{I}$ relative to $\mathbb{D}$ corresponding to the prefix of σ of length ℓ. □

If a trace is not free from partial-replacements, it may not be possible to obtain its solution via HTN planning (given the same inputs). A similar property exists in the CANPlan semantics: BDI-style recovery from failure enables solutions that cannot be found using CANPlan's built-in HTN planning construct.

Theorem 5.6. *There exists a domain $\mathbb{D}$, state $\mathcal{I}$, task network d, and successful execution trace $\mathcal{T}$ of d from $\mathcal{I}$ relative to $\mathbb{D}$ such that* $\mathsf{act}(\mathcal{T}) \notin \mathsf{sol}(d, \mathcal{I}, \mathbb{D})$.

Proof. Consider the trace from our running example, up to the point where an execution via partial-replacement is performed using method m_1. If the resulting task network is successfully executed, we get the solution corresponding to the sequence of action labels $8 \cdot 9 \cdot B \cdot 1 \cdot 4 \cdot 5 \cdot 3$, which is not an HTN solution; e.g. an HTN solution cannot have (actions corresponding to) both 8 and 1. □

6 AN ALGORITHM FOR HTN ACTING

In this section we present the Sense-Reason-Act algorithm for HTN acting, which combines our formalism with the processing of exogenous events. In the algorithm we use S_{nop} to denote the initial

[8]One could imagine a setting where *(i)* $\neg lowBat$ should only be checked immediately before action *mv*, and *(ii)* it is not undesirable to do actions *calib* (if $\neg cal$ holds) and *mvC*, even if action *mv* turns out to be non-applicable.

```
Sense-Reason-Act($\mathcal{I}, \mathbb{D}$)
1:  $d := [S_d, \phi_d] = [S_{nop}, true]$                       // Initial task network
2:  $\underline{\mathbf{T}} := S_{nop}$; $R := \{\langle S_{nop}, \emptyset\rangle\}$; $\overline{\mathcal{T}} := \langle d, \mathcal{I}, R, \mathbb{D}\rangle$
3:  while true do
4:      Set $\mathbf{T}$ to the possibly empty set of newly observed tasks
5:      $\mathbf{T}' := \{(n : t) \mid t \in \mathbf{T}, n$ is a unique task label$\}$
6:      $\underline{\mathbf{T}} := \underline{\mathbf{T}} \cup \mathbf{T}'$                  // Store all newly observed tasks
7:      $d' := [S_d \cup \mathbf{T}', \phi_d]$
8:      $R' := (R \setminus \{\langle \mathrm{top}(R), \emptyset\rangle\}) \cup \{\langle \mathrm{top}(R) \cup \mathbf{T}', \emptyset\rangle\}$
9:      if $\mathbf{T} \neq \emptyset$ then
10:         $\overline{\mathcal{T}} := \overline{\mathcal{T}} \cdot \langle d', \mathcal{I}, R', \mathbb{D}\rangle$
11:     end if
12:     if $\overline{\mathcal{T}}$ is neither successful nor blocked then // Below Def. 5.1
13:         Set $\langle d, \mathcal{I}, R, \mathbb{D}\rangle$ to an element of $\mathrm{exec}(d', \mathcal{I}, R', \mathbb{D})$
14:         $\overline{\mathcal{T}} := \overline{\mathcal{T}} \cdot \langle d, \mathcal{I}, R, \mathbb{D}\rangle$
15:     end if
16: end while
```

set of tasks $\{(0 : nop)\}$, and $\mathrm{top}(R)$ to denote the (unique) set S of tasks in the 'top level' reduction couple, given a set of reduction couples R, i.e., the couple $\langle S \supseteq S_{nop}, \emptyset\rangle$. The algorithm takes the current state and HTN domain as input and continuously performs two main steps as follows.

Step 1. The algorithm 'processes' newly observed (external) tasks (if any) and inserts them as top-level tasks to a copy of the current configuration's task network d and set of reduction couples R (lines 4 to 8), which are used to create the 'next' configuration.

Such tasks could be the initial requests, for example to transfer the soil data and sample and then recharge, or requests that arrive later, possibly while other tasks are being achieved. For example, task *procImg* could be a newly observed task in the iteration following the execution of the actions corresponding to task labels 8 and 9 in method m_4 (as opposed to *procImg* being an initial request). A newly observed task could also represent an exogenous event triggered by a change in the environment; e.g. the arrival of primitive task *stormy* could represent the event that it has just become stormy, and it could have the add-list $\{isStormy\}$, which will be applied to the agent's state when the task is executed. Given a domain $\mathbb{D} = \langle Op, Me\rangle$, we stipulate that any newly observed task t is such that $\mathrm{pre}(t, Op) = true$ if t is primitive, and $\mathrm{rel}(t, \mathbb{D}) \neq \emptyset$ otherwise.

Step 2. If one or more new tasks were indeed observed, the corresponding 'next' configuration is appended (line 10) to the current 'dynamic' execution trace, or *d-trace* $\overline{\mathcal{T}}$. A d-trace is slightly different to an execution trace (Definition 5.1) in that the former may include tasks that are not just obtained by reduction but also dynamically from the environment. If an execution via reduction, action, or replacement is possible from the last configuration in the d-trace (line 12), the execution is then performed and the resulting configuration is appended to the trace (lines 13 and 14).

The following theorem states that any d-trace produced by the algorithm is sound, i.e., any such d-trace, which may include new tasks observed over a number of iterations, is equivalent to some (standard) execution trace such that all of those tasks are present in the first configuration, but their execution is 'postponed'.

Theorem 6.1. *Let state $\mathcal{I}_{in}$ and domain $\mathbb{D}$ be the inputs of algorithm Sense-Reason-Act. Let $\underline{\mathbf{T}}_{16}$ and $\overline{\mathcal{T}}_{16}$ be the values of variables $\underline{\mathbf{T}}$ and $\overline{\mathcal{T}}$, respectively, on reaching line 16 in the algorithm (after one or more iterations). Then, $\mathrm{act}(\overline{\mathcal{T}}_{16}) = \mathrm{act}(\mathcal{T})$ for some execution trace $\mathcal{T}$ of task network $[\underline{\mathbf{T}}_{16}, true]$ from $\mathcal{I}_{in}$ relative to $\mathbb{D}$.*

Proof Sketch. D-trace $\overline{\mathcal{T}}$ in the algorithm, which is incrementally built, is similar to an execution trace, except for *(i)* the initial 'empty' task network of $\overline{\mathcal{T}}$; and *(ii)* the task networks appended in line 10 to account for newly observed tasks. We obtain an execution trace from $\overline{\mathcal{T}}_{16}$ as follows: take the last element $\tau_j \in \overline{\mathcal{T}}_{16}$ (each $\tau_k = \langle [S_k, \phi_k], \mathcal{I}_k, R_k, \mathbb{D}\rangle$) such that $S_j \subset S_{j+1}$, i.e., there are newly observed tasks in S_{j+1}; remove τ_{j+1} from $\overline{\mathcal{T}}_{16}$; add the elements in $S_{j+1} \setminus S_j$ to each S_i and $\mathrm{top}(R_i)$, for $i \in [1, j]$; and repeat these steps on the resulting d-traces until an execution trace is obtained. □

7 DISCUSSION AND FUTURE WORK

While some implementations of HTN acting frameworks do exist in the literature, this paper has, for the first time, provided a formal framework, by using the most general HTN planning syntax and building on the core of its semantics. In doing so, we have carried over some of the advantages of the HTN planning formalism, such as the ability to flexibly interleave the actions associated with a method [4], and to check a method's applicability *immediately* before first executing an action. We have also compared HTN acting to HTN planning, and to a BDI agent programming language.

We could now explore adding a 'controlled' and 'local' account of HTN planning into HTN acting. The result should be a similar semantics to CANPlan, which allows a BDI agent to perform HTN planning but only from user-specified points in a hierarchy. One approach might be, given a ground non-primitive task t, to use the construct $\mathrm{Plan}(t)$ to indicate that HTN planning (as opposed to an arbitrary reduction) must be performed on t, and to define the new notion 'execution via HTN planning'. Given a current configuration $\langle d, \mathcal{I}, R, \mathbb{D}\rangle$ with task network $d = [\{(n : \mathrm{Plan}(t)), \ldots\}, \phi]$, the definition would, for example, check whether there is a ground instance d'_t of a method-body $d_t \in \mathrm{rel}(t, \mathbb{D})$ such that $\mathrm{sol}(d'_t, \mathcal{I}, \mathbb{D}) \neq \emptyset$ holds (defined in Section 2).

We could also investigate an improved semantics where a 'tried' method-body is re-tried to achieve a task. Recall that when a relevant method-body is selected to reduce a task, and the body turns out to be 'non-applicable' (i.e., it is unable to execute any of its tasks) in the current state, we consider the body to have been 'tried', in the same way that we consider a body to have been tried if it fails (becomes blocked) *during* execution, e.g. due to an environmental change. To enable re-trying a body that was not applicable in an earlier state, we should at the least be able to check whether that state is different to the current one (both of which are sets of ground atoms). Ideally, however, we should also be able to quickly check (in polynomial time) whether the conditions that differ between the two states are likely to make the method-body applicable. To enable re-trying a method-body that had *failed*, we could explore techniques for analysing the conditions responsible for the failure in order to check that they no longer hold, as suggested in [11].

REFERENCES

[1] Ron Alford, Pascal Bercher, and David W. Aha. 2015. Tight Bounds for HTN Planning with Task Insertion. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1502–1508.

[2] Ron Alford, Vikas Shivashankar, Mark Roberts, Jeremy Frank, and David W. Aha. 2016. Hierarchical Planning: Relating Task and Goal Decomposition with Task Sharing. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 3022–3029.

[3] Giuseppe De Giacomo, Yves Lespérance, Hector J. Levesque, and Sebastian Sardina. 2009. IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents. In *Multi-Agent Programming: Languages, Platforms and Applications*, Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni (Eds.). Springer, Chapter 2, 31–72.

[4] Lavindra de Silva. 2017. BDI Agent Reasoning with Guidance from HTN Recipes. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*. 759–767.

[5] Lavindra de Silva and Lin Padgham. 2004. A Comparison of BDI Based Real-Time Reasoning and HTN Based Planning. In *Proceedings of the Australian Joint Conference on AI (AI)*. 1167–1173.

[6] Olivier Despouys and Francois Felix Ingrand. 1999. Propice-Plan: Toward a Unified Framework for Planning and Execution. In *Proceedings of the European Conference on Planning (ECP) (Lecture Notes in Computer Science (LNCS))*, Vol. 1809. Springer, 278–293.

[7] Kutluhan Erol, James Hendler, and Dana S. Nau. 1994. HTN Planning: Complexity and Expressivity. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 1123–1128.

[8] Kutluhan Erol, James A. Hendler, and Dana S. Nau. 1996. Complexity Results for HTN Planning. *Annals of Mathematics and Artificial Intelligence* 18, 1 (1996), 69–93.

[9] Thomas Geier and Pascal Bercher. 2011. On the Decidability of HTN Planning with Task Insertion. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1955–1961.

[10] Michael P. Georgeff and Francois Felix Ingrand. 1989. Decision making in an embedded reasoning system. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 972–978.

[11] Malik Ghallab, Dana Nau, and Paolo Traverso. 2016. *Automated Planning and Acting* (1st ed.). Cambridge University Press, New York, NY, USA.

[12] Malik Ghallab, Dana S. Nau, and Paolo Traverso. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers Inc.

[13] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer. 1999. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems* 2, 4 (1999), 357–401.

[14] Subbarao Kambhampati, Amol Dattatraya Mali, and Biplav Srivastava. 1998. Hybrid Planning for Partially Hierarchical Domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 882–888.

[15] John W. Lloyd. 1987. *Foundations of Logic Programming* (second ed.). Springer.

[16] Felipe Meneguzzi and Lavindra de Silva. 2015. Planning in BDI agents: A survey of the integration of planning algorithms and agent reasoning. *Knowledge Engineering Review* 30, 1 (2015), 1–44.

[17] Dana S. Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. 1999. SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 968–973.

[18] Dana S. Nau, Héctor Muñoz-Avila, Yue Cao, Amnon Lotem, and Steven Mitchell. 2001. Total-Order Planning with Partially Ordered Subtasks. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 425–430.

[19] Anand S. Rao. 1996. 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) (Lecture Notes in Computer Science (LNCS))*, Vol. 1038. Springer, 42–55.

[20] Anand S. Rao and Michael P. Georgeff. 1991. Modeling Rational Agents within a BDI-architecture. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 473–484.

[21] Sebastian Sardina, Lavindra de Silva, and Lin Padgham. 2006. Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*. 1001–1008.

[22] Sebastian Sardina and Lin Padgham. 2011. A BDI Agent Programming Language with Failure Recovery, Declarative Goals, and Planning. *Autonomous Agents and Multi-Agent Systems* 23, 1 (2011), 18–70.

[23] Vikas Shivashankar, Ron Alford, Ugur Kuter, and Dana Nau. 2013. The GoDeL Planning System: A More Perfect Union of Domain-independent and Hierarchical Planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2380–2386.

[24] Katia Sycara, Massimo Paolucci, Martin Van Velsen, and Joseph Giampapa. 2003. The RETSINA MAS Infrastructure. *Autonomous Agents and Multi-Agent Systems* 7, 1-2 (2003), 29–48.

[25] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. 1979. The Recognition of Series Parallel Digraphs. In *Proceedings of the Annual ACM Symposium on Theory of Computing*. 1–12.

[26] Michael Winikoff. 2005. Jack™ Intelligent Agents: An Industrial Strength Platform. In *Multi-Agent Programming: Languages, Platforms and Applications*, Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni (Eds.). Springer US, 175–193.

[27] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. 2002. Declarative & Procedural Goals in Intelligent Agent Systems. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 470–481.

[28] Zhanhao Xiao, Andreas Herzig, Laurent Perrussel, Hai Wan, and Xiaoheng Su. 2017. Hierarchical Task Network Planning with Task Insertion and State Constraints. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 4463–4469.