The HATP Hierarchical Planner: Formalisation and an Initial Study of its Usability and Practicality

Lavindra de Silva1 and Raphaël Lallement2 and Rachid Alami2

Abstract—HTN planners have generally relied on specialised languages for domain and problem representations. To facilitate adoption by other communities such as robotics, however, and integration with real world applications written in standard programming languages, we need HTN planners that are based on more familiar concepts from structured programming, and that come ready with features supporting integration. In this paper, we demonstrate how the HATP (Hierarchical Agent-based Task Planner) HTN planner offers such “syntactic sugar” and some of these features. Moreover, since it has a conceptually distinct syntax compared to traditional HTN planners, we also develop a formalism to unambiguously capture HATP’s syntax and an important subset of its semantics, which we then use to compare against the formalism of a well understood family of HTN planners and to show that the former is sound. Finally, we demonstrate that despite quite possibly using “heavier” data structures to naturally capture HATP’s syntax/semantics, and thereby facilitate extensions to HATP and integration with other applications, the implementation still performs acceptably.

I. INTRODUCTION

Hierarchical Task Network (HTN) planning is a proven approach to solving complex, real world planning problems more efficiently than planning from first principles when domain control knowledge is readily available [1], [2]. This information is embedded in the HTN domain as an intuitive hierarchy, which helps effectively guide the HTN search process towards the goal. By virtue of planning for tasks in the same order in which they are later executed, total-order HTN planners always know the complete state of the world at every planning step. This allows writing more expressive domain encodings than possible with partial-order HTN planning, such as preconditions having axioms (horn-clause inferencing) and calls to external procedures [1]. Such features have facilitated the use of total-order HTN planners in agent systems and seen them excel in AI games [3], [4], [5]. Nonetheless, these integrations have still relied on experts who understand the typically Lisp-like HTN representation and its specialised data structures. In this paper we study the HATP (Hierarchical Agent-based Task Planner) [6], [7] HTN planner, and show how it offers a conceptually different, more familiar representation than traditional HTN planners, while remaining sound and practical.

*We thank all the anonymous reviewers for their useful feedback. This work was partially supported by the European Commission under contract ICT 287617 (ARCAS).

1Institute for Advanced Manufacturing, University of Nottingham, Nottingham, UK, lavindra.desilvia@nottingham.ac.uk
2CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse, France & Univ. de Toulouse, LAAS, F-31400 Toulouse, France {raphael.lallement@,rachid.alami}@laas.fr

The planning component of RETSINA agents [8] was an early step towards such a planner. It offers a user-friendly modeling language for multi-agent planning domains, and supports integration with a variety of real world applications. The authors do not, however, explore the relation between RETSINA’s domain representation and semantics and the partial-order HTN planning approach [9] on which it is based. Like RETSINA, the Cascade language offers a user-friendly programming language syntax for software development, in particular, for describing data flow patterns to guide automated software composition. In [10], the authors propose a conversion from Cascade data flow patterns into the language of an extended HTN planner capable of planning with preferences. Our work, on the other hand, is the study of a self-contained HTN planning language and system, which can directly reason with its high-level constructs—they do not need to be translated into another language.

The need for non-specialised representations has also been recognised in Belief-Desire-Intention (BDI) agent systems, closely related to HTN planning in both syntax and semantics, and indeed even equivalent for the most part [11]. For example, the JACK [12] and JADE [13] systems offer a more familiar, Java-like representation and a corresponding execution engine, whereas PRS [14] and Jason [15] work with the conceptually different, more specialised Lisp-like and Prolog-like representations. On the other hand, while the latter two systems are more closely linked to the formal semantics of BDI agent systems, it is not obvious whether the user-friendly syntactic constructs of systems like JACK and JADE behave correctly with respect to their intended operational semantics. While HATP shares some of the syntactic advantages with systems like JACK and JADE, one goal of this paper is to also show that HATP’s syntax and an important subset of its corresponding semantics is sound.

The Python-based Pyhop1 total-order HTN planner is perhaps the most closely related system to ours. Unlike the representation used by its predecessors—the SHOP [1] family of planners (SHOP, SHOP2, ISHOP, and JSHOP2)—Pyhop allows encoding HTN domain elements as ordinary Python functions, and states as Python objects that contain variable bindings. Thus, Pyhop models First-Order Logic (FOL) literals as Python data structures and their addition and removal from a state as a modification to those data structures, allowing easier integration with applications such as AI games than its predecessors. However, as with JACK and JADE, the soundness and expressivity of Pyhop’s syntax

1https://bitbucket.org/dananau/pyhop
and semantics is yet to be studied. Pyhop’s approach shares similarities with the state variable representation, where predicates are replaced with an explicit functional notation allowing more concise domains in many cases [9]. While there are other planners such as AFSI [16] and IXTET [17] that are also based on the state variable representation, they do not follow the HTN approach.

The Hierarchical Agent-based Task Planner (HATP)—previously Human-Aware Task Planner—offers a conceptually rich domain representation that is the product of lessons learnt over years of practical integration with real robotic systems and other applications. HATP has been extended in various ways: including support for splitting a solution into sub-solutions and assigning them to the various agents in the domain [18]; modelling their beliefs as distinct world states [19]; allowing “social rules” to be added by the user to define what kind of robot/agent behaviour is appropriate [6], [18]; allowing tasks to be planned by taking the human’s safety and comfort into account [20]; and to interleave HTN planning with geometric reasoning [21], [22]. However, none of these works have studied HATP’s “syntactic sugar” and associated semantics, shown that they are sound, nor explored the existing implementation’s practicality, i.e. its runtime performance and what features facilitate integration with applications in multi-agent systems and robotics.

Thus, the contributions of this paper are threefold. First, we demonstrate how HATP offers syntactic sugar over traditional HTN representations, and certain features to facilitate integration with real world applications. To this end we illustrate HATP code from our own encoding of a known planning domain. Second, since HATP’s syntax is conceptually distinct from that of traditional HTN planners, we develop an unambiguous specification for it, concretise the relation between the syntax and semantics of HATP and a well understood family of HTN planners, and then show that the former is sound. In particular, we develop a formalism that captures the syntactic constructs of HATP and the important aspects of its semantics: specifically, what it means to apply an effect to a state and for a precondition to hold in it. While HATP’s syntax and semantics have been naturally realised as typical object oriented data structures, and thereby facilitated extensions to HATP and integration with other applications, these data structures seem to be “heavier” than the more compact, list-based data structures used in traditional HTN planning systems. Thus, after briefly reviewing certain implementation decisions, our third contribution is to show empirically that the implementation performs acceptably.

II. BACKGROUND

HTN planning

While first principles planners such as STRIPS focus on achieving a goal state, HTN planners focus on solving “abstract tasks”. This paper deals only with total-order HTN planning [1]. A (total-order) HTN planning problem, then, is the 3-tuple \((d, s_0, D)\), where \(d\), the “goal” to achieve, is a sequence of primitive/abstract tasks, \(s_0\) is the initial state, and \(D\) is an HTN planning domain. The latter is the pair \((A, M)\) where \(A\) is a finite set of operators and \(M\) a finite set of methods. An operator is as in first principles planning, an action is a ground instance of an operator, and a primitive task is simply the name of an operator/action. A method is a 4-tuple consisting of: the name of the method, the abstract task (a name) that it needs to solve, a precondition specifying when the method is applicable, and a body realising the “decomposition” of the task associated with the method into more specific subtasks. Specifically, the method-body is a sequence of primitive/abstract tasks.

The HTN planning algorithm selects applicable methods from \(M\) and applies them to associated abstract tasks in \(d\) in a depth-first manner. In each iteration this will typically result in \(d\) becoming a “less abstract” task sequence. This process repeats until \(d\) has only primitive tasks left. At any stage during planning if no applicable method can be found for an abstract task, the planner essentially “backtracks” to try an alternative method for an abstract task refined earlier.

Preliminary notation

In the formalisms that follow, our language has the following (possibly infinite) disjoint sets of 0-valence non-logical symbols (letters): the set \(V\) representing variables; \(P\) predicates; \(F\) functions, representing evaluable predicates; \(P_{op}\) primitive tasks; \(P_m\) methods; \(P_{sk}\) abstract tasks; \(T_e\) entity types; \(T_b\) the basic data types float, boolean and string, respectively; \(O_e\) entity objects; and \(O_s\) the set of all 0-valence non-logical symbols not in the other sets. We use \(T\) to denote \(T_e \cup T_b\). Our language also has the set \(Q\) of rational numbers and the symbol \(\lambda\) which denotes the empty string.

We assume that there is a surjective function \(m_e : O_e \rightarrow T_e\) from entity objects to entity types (possibly with multiple entity objects mapping to a single entity type). We define the more general mapping function \(m = m_e \cup \{(s, st) \mid s \in O_s\} \cup \{\top, \bot, (\land, \lor) \cup \{(n, fl) \mid n \in Q\} \cup \{\lambda\}\}, \) where \(\land\) and \(\top\) are the trivial propositions. Moreover, given any type \(t \in T\) we define its objects as \(O(t) = \{e \mid m(e) = t\}\) and the set of basic objects as \(O_b(t) = O(fl) \cup O(b) \cup O(st)\).

In the formalisms in this paper, we often write the tuple \((e_1, \ldots, e_n)\) as \(e_1 \ldots | e_n\) for readability (when there are too many parentheses elsewhere), and we use \(|S|\) to denote the cardinality of set \(S\) and \(e\) the string with no symbols. Finally, we use Prolog conventions: predicate symbols and constants begin with lower case, and variables with upper case.

III. HATP LANGUAGE SYNTAX

A. HATP data structures and states

As a running example in this paper we show HATP code from our own encoding of the Dock Worker Robots (DWR) domain [9]. In this domain there are robots that carry containers, and crane robots that lift and put down containers from/onto piles. Piles store containers and are attached to locations. The goal is to move containers between piles by using robots and cranes.

The HATP planning domain and problem rely on a separately defined collection of data structures called “HATP
define entityType
Loc, Pile, Con, Kinds;
define entityAttributes Loc {
  static set Loc adjacent;
  dynamic atom boolean used;
}
define entityAttributes Pile {
  static atom Loc attached;
  dynamic set Con stores;
  dynamic atom Con top;
}
define entityAttributes Con {
  dynamic atom Pile at;
  dynamic atom Con on;
}
define entityAttributes Kinds {
  static atom string ROBOT;
  static atom string CRANE;
}
define entityAttributes Agent {
  // an agent is a robot or crane
  static atom string kind;
  // for robots
  static atom Loc attached;
  // for robots and cranes
  dynamic atom Loc at;
  // for robots and cranes
  dynamic set Path loc;
  // for robots
  dynamic atom Con carry;
  // for cranes
  dynamic atom Con on;
  // for robots and cranes
  dynamic atom Con store;
}

define entityType
Kinds;
new Pile;
new Kinds;
new Con;
new Agent;
new Robot;
new Crane;

Fig. 1: HATP data structures (left to right) for the DWR domain. There are five entity types: Agent (default entity), which can be a robot or crane; Kinds; Loc (Location); Pile; and Con (Container).

"entities", which are the agent types and object types in the world. In HATP, the distinction between agents and other objects is important. Agents are treated as "first class citizens", and different types of agents may be defined by declaring different attributes within the Agent entity. Moreover, each operator is required to indicate which agent type(s) are associated with it. Agent objects of these agent types are used by extensions of HATP [19], [18] to, for example, split the final plan into a network of synchronised subplans linked to the different agent objects; the subplans may then be executed concurrently in a multi-agent system by synchronising as necessary.

An entity has a set of attributes. Each of these represent either a single data value, a set of data values, or a relation between the entity and other entities. Figure 1 shows an example of entities and attributes in HATP code. In order to be unambiguous about the syntax of HATP, and to study its properties, we shall now formalise the HATP data structures described.

An HATP attribute a is the tuple (p,t) where p ∈ P is a (0-valence) predicate and t ∈ T is an entity/basic type. An HATP entity e is a tuple (tₑ, {a₁,...,aₙ},dyn,set) where the following hold: n ≥ 0; tₑ ∈ Tₑ is an entity type; each aᵢ is an HATP attribute; for all aᵢ, aⱼ, if aᵢ ≠ aⱼ then pᵢ ≠ pⱼ (aᵢ = (pᵢ,tᵢ) for any k); and dyn, set are functions from {a₁,...,aₙ} to {T,⊥} that indicate respectively whether an attribute a in the set is dynamic (dyn(a) = T) or static (dyn(a) = ⊥), and whether a can have multiple elements (set(a) = T) or just one (set(a) = ⊥). An HATP entity library E₁ is a set of HATP entities with no two entities in it having the same first element (entity type). For example, the first entity in figure 1 can be formalised as

(loc,
{(adjacent,loc),(used,bl)},
{{(adjacent,loc),⊥},((used,bl),T)},{(adjacent,loc),T},{{(used,bl),⊥}}
).

\[
\text{Fig. 2: Part of the DWR HATP initial state}
\]

Figure 2 shows how an HATP initial world state is created by instantiating entities and assigning values to their attributes. Observe that, like the entity and attribute definitions in figure 1, the initial state specification also follows a familiar syntax. We formalise the latter as follows. Given an entity \( (t,\{a₁,...,aₙ\},\text{dyn, set}) \), an entity-attribute instance of one of its attributes \( aᵢ = (pᵢ,tᵢ') \) is any tuple \( (c,p,c') \) such that \( c ∈ O(t) \) and \( c' ∈ O(t') \). Thus, an HATP state \( S_h \) is a set of entity-attribute instances (of attributes of one or more entities), and an HATP initial state is one such state. For example, we could formally encode part of the HATP state in figure 2 as

\[
S_h = \{ (\text{crane1.kind, crn}),(\text{loc1, used, T}),\text{(robot, at, loc1),(crane1, attached, loc1)}\},\text{(pile1, stores, con1),(pile1, stores, con2)}\}.
\]

It is not difficult to see how the rest of the state could be formally encoded; the main subtlety is that \( S_h \) will not include tuples for attributes that have the value NULL.

B. HATP planning domains

Like in standard HTN planning, an HATP domain consists of a set of methods and a set of operators. For defining these, however, HATP employs some useful structured programming concepts such as control structures, blocks, typing, and subroutines. This allows, for instance, performing set operations on attributes that represent sets of values, including checking the size of a set. Moreover, unlike other HTN planners, elements in HATP preconditions and effects are evaluated and applied, respectively, in the order they are written—i.e. symbol ; denotes a sequence. This is especially important when using evaluable predicates that invoke computationally expensive subroutines: such predicates can appear later in the precondition/effect to avoid evaluating them unnecessarily. For example, consider the operator in figure 3. We could imagine a new line of code "canMove(Fm,70) == true;" as the precondition’s last line, where evaluable predicate canMove checks using a motion planner and a 3D world model whether there exists a trajectory between the two locations.

Taking inspiration from figure 3 we shall now start formalising HATP planning domains. An HATP action is a ground instance of an HATP operator, which is the tuple \( (\text{op}(\vec{v}),\Phi,Ξ) \). Element \( \text{op}(\vec{v}) \) is the name of the operator, where \( \text{op} ∈ P_{\text{op}} \) and \( \vec{v} = v₁ : t₁,...,vₙ : tₙ \) is a vector of typed distinct variables with each \( tᵢ ∈ Tₑ \). Element \( Φ, Ξ \) the
Fig. 3: The listing shows an HATP operator for the DWR domain which keeps track of the path travelled. The user-defined function \( \text{costToMove}(\text{Fm}, \text{To}) \) returns the estimated cost of moving from \( \text{Fm} \) to \( \text{To} \); "rem" stands for remove.

operator's precondition, is defined by the grammar

\[
\Phi ::= \Psi \mid \Phi ; \Phi \mid \forall (v : t), \Psi \rightarrow \Psi \mid \exists (v : t), \Psi \rightarrow \Psi
\]

\[
\Psi ::= C \mid C = \neg C \mid \Psi \cdot \Psi
\]

\[
C ::= t \in T_e, v \in V, p \in P, f \in F, c \in O_h, b \in O(b) \quad \text{(recall that these are respectively sets of entity types, variables, predicates, functions, basic objects, and booleans)}
\]

\[
\text{where } t \in T_e, v \in V, p \in P, f \in F, c \in O_h, b \in O(b)
\]

Next, we define the effect of an operator. Formally, an element \( \Xi \), the operator's effect, is defined by the grammar

\[
\Xi ::= E \mid \Psi ? E | \forall (v : t), \Psi ? E | \Xi ; \Xi
\]

\[
E ::= (v, p) \ll \tau | (v, p) \gg \tau | (v, p) \ll \tau | \exists (v, p) | \ll \lambda | E
\]

\[
\text{where } p \in P, v \in V \text{ and } \tau \text{ is as defined earlier. We call } E \text{ an effect atom/literal. Symbols } \gg \text{ and } \ll \text{ denote respectively removing and adding an element from/to a set; } \ll \text{ denotes assigning a value to an attribute; and } \Psi ? E \text{ denotes a guarded sub-effect, which is read "if } \Psi \text{ holds then do } E." ^2 \text{ For example, the conditional effect } \Xi \text{ below shows a possible formal encoding of the "effects" segment in figure 3:}
\]

\[
\Xi = (R, \text{at}) \ll \text{To} : (\text{Fm, used}) \ll \perp ;
\]

\[
\forall (L, \text{loc}), \Psi ? E ; \ldots ;
\]

\[
\text{where } \Psi = L \in \{ R, \text{path} \} : \text{To} = \text{Dest}, \text{ and }
\]

\[
E = (R, \text{path}) \gg L.
\]

\[\text{method Transport(Kinds K, Con C, Pile Target) \{}
\]

\[
\text{empty}(\text{Cat} \Rightarrow \text{Target}) ; \quad \text{// do nothing if container is at target}
\]

\[
\text{// container’s pile and target pile are at different locations}
\]

\[
\text{preconditions} \{ \text{Cat} \cdot \text{attached} \Rightarrow \text{Target} \cdot \text{attached} \};
\]

\[
\text{subtasks} \{
\]

\[
\text{Cr1} = \text{SELECT}(\text{Agent},
\]

\[
\{ \text{Cr1.kind} \Rightarrow \text{Kinds.CRANE} ;
\]

\[
\text{Cr1.\text{attached} \Rightarrow \text{Cat} \cdot \text{attached} \};
\]

\[
\text{Cr2} = \text{SELECT}(\text{Agent},
\]

\[
\{ \text{Cr2.kind} \Rightarrow \text{Kinds.CRANE} ;
\]

\[
\text{Cr2.\text{attached} \Rightarrow \text{Target} \cdot \text{attached} \};
\]

\[
\text{R} = \text{SELECTORDERED}(\text{Agent},
\]

\[
\{ \text{R.kind} \Rightarrow \text{Kinds.ROBOT} ;
\]

\[
\text{distance} (\text{R.at, Cat} \cdot \text{attached}) \ll ;
\]

\[
1: \text{GetReady}(\text{R, C, Cat});
\]

\[
2: \text{LoadRobot(Cr1, R, C)} > 1;
\]

\[
3: \text{NavFromTo(R, Cat\cdot\text{attached}, Target\cdot\text{attached})} \ll 2;
\]

\[
4: \text{UnloadRobot(Cr2, R, C)} > 3;
\]

\[
5: \text{Put(Cr2, C, Target)} > 4;
\]

\[
\}
\]

\[
\}
\]

\[
\}
\]

**Footnote:**

\[\text{For brevity we (WLOG) omit } \Xi ::= \Psi_1 ? \Psi_2 ? E \text{ as its RHS is semantically equivalent to } \forall (v : t), (\Psi_1 \land \Psi_2) ? E.\]

\[\text{The second element of an HATP domain is a set of methods. The body of a method is a sequence of tasks with values for their variables possibly selected from within the method-body itself. Specifically, declaring variables in HATP methods and controlling their bindings is done via the intuitive constructs SELECT, SELECTORDERED, and SELECTONCE, two of which are illustrated in figure 4. One might have noticed from the figure that a method allows subtasks to be partially ordered, achieved by not specifying ordering constraints between tasks in method bodies; e.g., removing constraint "\( > 2 \)" from the method in figure 4 would then not require the task labeled 3 to occur after the one labeled 2. This differs from the ability to interleave the steps belonging to two or more methods, as offered in SHOP2 [2]. HATP’s partial ordering feature, however, is merely a shortcut to supplying multiple totally ordered methods corresponding to every linearisation (total ordering) of the partially ordered subtasks. This is exactly what is done during planning: any partially ordered subtasks in a body are handled by taking all their linearisations, thereby creating additional method options to consider for the parent task’s decomposition. To keep the formalisms simple, we assume that method bodies are totally ordered. Moreover, since the constructs SELECTONCE and SELECTORDERED may only offer an improvement in planning efficiency or solution cost compared to SELECT, it is not interesting to formalise them. Then, an HATP method is defined as a tuple \( (m(\vec{v}), \text{tsk}(\vec{v}'), \Phi, \text{body}) \), where \( m(\vec{v}) \) is the method’s name and \( \vec{v} = t_1 : v_1, \ldots, t_n : v_n \) is defined as before; \( \text{tsk}(\vec{v}') \) is the abstract task that this method handles where \( \text{tsk} \in P_\text{tsk} \) and \( \vec{v}' \) is a subsequence of \( \vec{v} \); \( \Phi \), the precondition of \( m(\vec{v}) \) is defined as before; and \text{body} is a possibly empty sequence of tasks where each task...**
$\text{tsk}'(v_1', \ldots, v_m')$ is such that $\text{tsk}' \in P_{\text{tsk}} \cup P_{\text{op}}$ and each $v_i' \in \{v_1, \ldots, v_n\}$. Note that the functionality of the SELECT construct is accounted for implicitly in this definition.

Well-defined HATP domains: A precondition/effect derived from the grammars above will not necessarily conform to the correct use of types and operations. In the effect $\Xi$ from before, for example, it would not be correct to have $\Psi = \text{To} \in \text{Dest} ; R \in (\text{R}, \text{path})$ or $E = (\text{R}, \text{loes}) \gg L$, as $\text{Dest}$ is not a set, our agents do not have a $\text{loes}$ attribute, and $R$ is not a location whereas $\text{path}$ is a set of locations. Hence, we must define what constitutes a meaningful derivation. The first definition requires HATP domains to be type-consistent. Basically, an operator/method is type-consistent if for any precondition or effect literal mentioned in it, any variable occurring in the literal has the right scope, and the LHS (left-hand side) and RHS (right-hand side) of the literal’s atom involves the same type.

Definition 1: (CONSISTENT TYPES) Let $\text{con}$ be any conjunct of the precondition or effect (if any) of any operator or method named $\mathbf{h}(\mathcal{v})$, and $l$ any precondition or effect literal mentioned in $\text{con}$. If $V_n = \{v : t \mid \mathcal{v}$ mentions $v : t\}$, the scope of $l$ in $\text{con}$, denoted $\text{scope}(l, \text{con})$, is $V_n \cup \{v : t\}$ if $\mathcal{v}$ mentions $v : t$, and $V_n$ otherwise. If $lh$ is the LHS of the atom of $l$ and $rh$ its RHS, $\mathbf{h}(\mathcal{v})$ is type-consistent relative to an entity library $\mathcal{E}l$ if $rh = \lambda$ or $\text{type}(lh) = \text{type}(rh)$, with

$$\text{type}(\tau) = \begin{cases} \tau' & \text{if } \tau = \mathbf{O}(\tau') \wedge \tau' \in \{f, t, b\}; \\
fl & \text{if } \tau = \mathbf{L}(v, p) \wedge (v : t) \in \text{scope}(l, \text{con}) \wedge \\
\tau & \text{if } \tau = \mathbf{L}(v, p) \wedge (v : t) \in \text{scope}(l, \text{con}) \wedge \\
\{t, A, \text{dyn, set}\} \in \mathcal{E}l \wedge (p, \tau') \in \mathcal{A}; \\
t & \text{if } (\tau : t) \in \text{scope}(l, \text{con}); \text{ and} \\
b & \text{if } \tau = f(x_1, \ldots, x_n). \end{cases}$$

Similarly, we can, for example, easily define an entity library as operation-consistent if static attributes are not modified in effects, and set operations are only performed on sets and assignment operations only on attributes that are not sets. In the next sections, we assume that our entity libraries and operator libraries are consistent as defined by these notions.

C. Link with traditional HTN planners

To put HATP’s conceptually distinct representation into perspective, we now show how it is related to the syntax of the SHOP family of planners—the dominant total-order HTN planners in the literature. SHOP’s syntax also shares similarities with PDDL and hence other planners in the literature. In the mapping that follows, we slightly deviate from SHOP’s syntax for readability, and from now on we shall only focus on non-trivial HATP elements.

We start by mapping HATP preconditions into SHOP preconditions. As hinted in figure 2, the main idea is that attributes of HATP entities map to SHOP predicate symbols, and HATP entities and attribute values to the parameters of SHOP predicates. Translating the HATP language construct

$?(v, p)$—which checks the size of entity $v$’s (e.g. $\text{Pile}$’s) attribute $p$ (e.g. $\text{stores}$) representing a set of values—requires special care, as it amounts to the “meta-level” functionality of determining the number of relevant ground atoms in the SHOP state. To this end, as we demonstrate concretely below, we use a special SHOP predicate symbol $(s)$ associated with the number, having $p$ and $v$ as the first two parameters. Formally, then, the SHOP precondition corresponding to an HATP precondition $\Phi$ is $\mathbf{P}(\Phi) =$

$$\mathbf{P}(\Phi_1) \land \mathbf{P}(\Phi_2)$$

if $\Phi = \Phi_1; \Phi_2$ and $\Phi_1, \Phi_2 \neq c$; 

forall($v$; $t(v)$ \& $\mathbf{P}(\Phi)$, $\mathbf{P}(\Phi')$) 

$p(v, v_3) \land p_2(v_2, v_3)$ 

if $\Phi = \forall(v : t), \Psi \rightarrow \Psi'$; 

if $\Phi = \forall(v : t)p \in v_2[p_2]$ or $v[p] = v_2[p_2]$; 

forall($v_3, v_4$) $p(v, v_3) \land p_2(v_2, v_4), (v_3 \neq v_4)$ 

if $\Phi = \neg \forall(v : t)p \in v_2[p_2]$; 

$p(v', v)$ 

if $\Phi = v \in v'[p]$; 

$p(v_2[p_2] \leq p)$ 

if $\Phi = \neg \forall(v : t)p < v_2[p_2]$; 

$p(v_1, v_2 \land p_2(v_2, v'), (v' \leq v')$ 

if $\Phi = \forall(v : t)p \in v_2[p_2]$; 

$p(v, v_3) \lor p_2(v_2, v_4)$ 

if $\Phi = \forall(v : t)p \in v_2[p_2]$; 

$p(v, v') \land (c \leq v')$ 

if $\Phi = c \leq v[p]$; and 

undefined 

otherwise.\(^4\)

In words, checking $v[p] \in v_2[p_2]$—whether the value associated with attribute $p$ of entity $v$ is in the set corresponding to attribute $p_2$ of entity $v_2$—amounts to taking the value $v_3$ associated with the SHOP predicate $p$ having $v$ as its first parameter, and then checking if $v_3$ is also one of the values associated with the predicate $p_2$ having $v_2$ as its first parameter; conversely, the opposite is done for translating the effect literal $\neg(v[p] \in v_2[p_2])$. Similarly, checking $v_1[p_1] \leq v_2[p_2]$ amounts to determining whether $v \leq v'$ holds for the numbers $v, v'$ associated with SHOP predicates $p_1, p_2$ having entities $v_1, v_2$ as first parameters. Finally, universal quantification in HATP amounts to explicitly checking in SHOP that variable $v$ is of the correct type, which is achieved via the SHOP predicate with symbol $t$.

Next, we show how an HATP effect is mapped to an equivalent SHOP effect. Formally, the SHOP effect corresponding to an HATP effect $\Xi$ is $\Xi(\Xi) =$

$$\Xi(\Xi_1) \land \Xi(\Xi_2)$$

if $\Xi = \Xi_1; \Xi_2$ with $\Xi_1, \Xi_2 \neq c$; 

if $\mathbf{P}(\Psi), \mathbf{E}(E)$ 

if $\Xi = \Psi?E$; 

forall($v$; $t(v) \land \mathbf{P}(\Psi), \mathbf{E}(E)$) 

$\neg p(v, \tau)$ 

if $\Xi = \forall(v : t), \Psi?E$; 

if $\Xi = \forall(v, \tau) \gg \tau$; 

if $\Xi = \forall(v, \tau) \gg \tau$; 

$\neg p(v, \tau)$ 

if $\Xi = \forall(v, \tau) \gg \tau$; 

if $\Xi = \forall(v, \tau) \gg \tau$; 

$\neg p(v, v')$ 

if $\Xi = \forall(v, \tau) \gg \tau$; and 

undefined 

otherwise.\(^5\)

In words, adding a value $\tau$ to the set corresponding to attribute $p$ of entity $v$, i.e. operation $(v, p) \ll \tau$, simply amounts to adding the atom $p(v, \tau)$ to the SHOP state; operation $(v, p) \gg \tau$ is analogous. However, assigning a

\(^4\)Variables $v', v_3, v_4$ are unique. Recall that we leave out the obvious cases such as $\exists(v : t), \Psi \rightarrow \Psi'$ and $v[p] < c$, and that we write $(e_1, \ldots, e_n)$ as $e_1|\ldots|e_n$ for readability.

\(^5\)Variable $v'$ is unique, and if is a special case of forall.
value \( \tau \) to an HATP attribute \( p \) of entity \( v \), i.e. operation \((v, p) \leftarrow \tau \), amounts to more conjuncts in SHOP, which first checks if its associated SHOP atom exists, and if so, “updates” its second parameter’s value by assigning it the value \( \tau \). Likewise, setting an attribute to the empty string, i.e. operation \((v, p) \leftarrow \lambda \) amounts to checking the corresponding value \( v' \) in SHOP and removing the associated atom from the state.

Thus, apart from being intuitive, the other advantage of the HATP representation is akin to the advantage the state-variable representation over the classical [9]. Specifically, some HATP effects are more “concise” than the corresponding SHOP ones in the sense that certain single HATP effect constructs map to SHOP expressions having multiple conjuncts. The same applies to certain single HATP precondition constructs. This is exemplified by the following operator of a DWR domain variant, where robots can carry a pile of containers (instead of just one), and we need to be able to shift containers \( C_1 \) from the top of a robot’s pile \( P_1 \) to the top of a pile \( P_2 \) on the ground, and vice versa, without ever storing more than 3 containers on a pile.

<table>
<thead>
<tr>
<th>SHOP</th>
<th>HATP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precond.</td>
<td>( \top(P_1, C_1) \land \top(P_2, C_2) \land ) ( \text{size}(\text{stores}, P_2, S) \land S &lt; 3 )</td>
</tr>
<tr>
<td>Effect</td>
<td>( \top(P_1, C_3) \land \top(P_2, C_1) \land \neg \top(P_1, C_1) \land \neg \top(P_2, C_2) )</td>
</tr>
</tbody>
</table>

Notice that the SHOP operator is more concise in the syntax of HATP (the corresponding HATP code looks similar). We have observed from a preliminary study that in practice, e.g. in the three domains that we have used for the empirical evaluation, HATP preconditions and effects need less conjuncts on average than their SHOP counterparts. We leave a thorough exploration of this relation to future work.

So far we have shown how HATP preconditions and effects map to their SHOP counterparts. To conclude this section, we show how an HATP state maps to a SHOP state. Formally, the SHOP state \( S \) corresponding to an HATP state \( S_h \) is \( \bigcup_{(c,p,c_1) \in S_h} \{ (t(c_1, p(c_1), s(p, c_1, c_2 : (c, p, c_2) \in S_h)) \} \} \), where \( t = m_s(c) \) is the type of \( c \).^7

IV. HATP LANGUAGE SEMANTICS

Given an HATP domain, entity library, and state, we now describe how parts of these elements are evaluated/manipulated as part of the planning process. Recall that this essentially involves repeatedly applying methods and actions—if their preconditions hold—to respectively the pursued partial plan and world state, until no abstract tasks remain. First, we formally define how preconditions are evaluated and effects are applied, and we then show that these semantics are sound with respect to the semantics of SHOP.

To determine if an operator or method is applicable in a given HATP state, and indeed also whether guarded sub-effects occurring in operators can be applied to a state, we must define how a given HATP precondition—having individual elements—is evaluated relative to the state. Formally, if \( S_h \) is an HATP state, a ground HATP precondition \( \Phi \) (i.e. where no free variables occur) is evaluated as follows. A precondition \( C \) mentioned in \( \Phi \) evaluates to \( \top \) if \( C \) is

- \( c \in c_2[p] \) or \( c_2[p] = c \)
- \( \neg \exists c \left( (c_1, p) \in S_h \right) \)
- \( \neg \exists (c_2[p], p) \land \neg \exists c \left( (c_2[p], c_3) \in S_h \right) \)
- \( \neg \exists c \left( (c_2[p], c_3) \in S_h \right) \)
- \( \neg \exists c \left( (c_2[p], c_3) \in S_h \right) \)

otherwise, \( C \) evaluates to \( \bot \). Connectives \( \neg \), \( \top \) occurring in \( \Phi \) are evaluated as in FOL, the sequence relation : as connective \( \land \) in FOL, and quantifiers \( \forall \) and \( \exists \) are treated as \( \forall v \in O(t) \) and \( \exists v \in O(t) \) in FOL, respectively.

Then, an HATP precondition \( \Phi' \) holds in \( S_h \), or \( S_h \models \Phi' \), if a ground instance of \( \Phi' \) evaluates to \( \top \) relative to \( S_h \).

Next, we define what it means for an HATP effect to be applied to a given HATP state. If \( S_h \) is a HATP state and \( \Xi \) a ground HATP effect, the result of applying \( \Xi \) to \( S_h \) is defined as \( R_h(\Xi, S_h) = \)

\[
\begin{align*}
(\Xi) & = (c, p) \iff \lambda \land \not\exists v \cdot p(c) \in S_h; \\
R_h(\Xi_1, \Xi_2) & = (\Xi_1 : \Xi_2 \in \Xi_2 \neq \epsilon; \\
R_h(\Xi \theta_1, \dotsc, \theta_n, S_h) & = \forall v : t, \Xi' \land \Xi \iff \Xi \land \exists v : t, \Xi' \land \\
\text{with each } \theta_i & = \{v/c_i\} \\
R_h(E, S_h) & = \Xi \iff \not\Xi \land \Xi \neq \Psi; \\
S_h & = \Xi \iff \not\Xi \land \Xi \neq \Psi; \\
S_h \setminus \{c[p']\} & = \Xi \iff \Xi \land (c, p) \not\exists c'; \\
S_h \cup \{c[p']\} & = \Xi \iff \Xi \land (c, p) \not\exists c'; \\
S_h \setminus \{c[p']\} & = \Xi \iff \Xi \land \exists c[p'] \in S_h; \\
S_h \cup \{c[p']\} & = \Xi \iff \Xi \land \exists c[p'] \in S_h; \\
S_h \setminus \{c[p']\} & = \Xi \iff \Xi \land \exists c[p'] \in S_h; \\
\text{undefined} & \text{otherwise.}
\end{align*}
\]

Basically, the result of applying an effect to a state is the result of recursively applying each of its individual elements to the original state or the intermediate ones. If an element is a universal quantifier over another effect, all of its relevant ground instances are applied to the state, and if it is a guarded sub-effect \( \Psi \land \Xi \), effect \( \Xi \) is applied only if \( \Psi \) holds in the (possibly intermediate) state \( S_h \). Observe that the elements of an effect are applied in the same order that they appear in the effect. Such ordering can be advantageous as it removes the need to re-test an expensive evaluative predicate—e.g. canMove(Fm,To)—occurring in a guarded sub-effect if it was already tested in an earlier such step. For example, canMove(Fm,To) could be separately tested at the beginning of the operator’s effect; \( \top \) assigned to a boolean attribute (with

^6We acknowledge that another possible notion of conciseness is the length of a precondition/effect.

^7Actually, \( S \) also has members for HATP sets of size 0, and all relevant \( s(p, c, n) \in S \) will be updated with the \( \ll \) and \( \gg \) set operations. Recall that \( m_s(c) \) is a function from entity objects to entity types, and that symbol \( s \) represents the size of a set.

^8Each \( \Xi \theta_i \) is “coherent”—it does not add as well as remove a given \( c_j \).
an initial value of \( \bot \)) if the evaluable predicate holds; and the boolean attribute used in place of the predicate in subsequent steps. Semantically, however, we expect an HATP effect to be “deterministic”, i.e. the result of applying it should not depend on the order of its steps. This assumption is stated formally below.

**Definition 2:** (Deterministic Operators) An HATP operator is said to be deterministic if for all its ground instances \((\text{op}, \Phi, \Xi)\), HATP states \(S_h\), and \(\Xi'\) obtained from \(\Xi\) by changing the order of conjuncts mentioned in \(\Xi\) (including those in quantifiers), if \(S_h \models \Phi\) then \(R_h(\Xi, S_h) = R_h(\Xi', S_h)\).

We can now state that the presented HATP semantics and mapping from HATP to SHOP is sound. The first theorem states that an HATP method/action precondition holds in an HATP state if and only if the corresponding SHOP precondition holds in the corresponding SHOP state.

**Theorem 1:** Let \(S_h\) be an HATP state, \(S\) its corresponding SHOP state, and \(\Phi\) an HATP precondition. Then, \(S_h \models \Phi\) if and only if \(S \models \mathcal{P}(\Phi)\).

The second theorem states that if an HATP effect is applied to an HATP state \(S_h\), the resulting state is equal (after mapping it to the SHOP counterpart) to the one we get by applying the corresponding SHOP effect to the SHOP state that maps to \(S_h\). Function \(R(\phi, S)\) below denotes the result of applying a SHOP effect \(\phi\) to a SHOP state \(S\).

**Theorem 2:** Let \(S_h\) be an HATP state, \(S\) its corresponding SHOP state, \(\Xi\) a ground HATP effect, and \(S'\) the SHOP state corresponding to \(R_h(\Xi, S_h)\). Then \(S' = R(\Xi(S_h), S)\).

Using these theorems and the fact that the algorithm that HATP is based on that of SHOP [6], it is not difficult to show that SHOP will produce the same plans as HATP, given any HATP planning problem and its mapping to SHOP.

**V. Implementation Issues and Performance**

By virtue of being based on structured programming concepts, HATP’s syntax and semantics lend themselves to being implemented directly using “raw” C++ data structures such as classes and complex data types. This has facilitated various extensions to HATP as outlined in the introduction, and integrations with other applications written in standard programming languages; indeed, these are also some of the main advantages of the Python-based Pyhop HTN planner. More specifically, unlike other HTN planners which model a state as a set of FOL atoms, an HATP state is implemented as a data structure representing a set of ground HATP entities. This design differs only slightly from our definition of an HATP state as a set of entity-attribute instances, and lends itself to implementing a state as a set of C++ objects instantiating the C++ classes that represent HATP entities; figure 2 shows 10 such C++ objects. From within evaluable predicates, this design allows convenient access to data organised as C++ objects, which is an important feature for integrating HATP with other systems, as done for instance in [21], [22].

However, this design choice also seems to make HATP partial plans and states less compact, and require more space than those of the SHOP family of planners, which use traditional, list-based data structures. In particular, HATP uses certain internal “administrative” data structures which are inherited by C++ classes of entities: e.g. a parent class reserves a possibly unused set—of pointers to attribute names—for each attribute type (e.g. a string) to cater for any attribute type that might be needed by inherited entities. Combined with the fact that HATP code is not optimised, its performance does not, as expected, compare with the fastest HTN planners: SHOP, SHOP2 and JSHOP2. Thus, to show that our current implementation of HATP still performs acceptably, we use the JSHOP planner—the Java version of SHOP—as a baseline; JSHOP has already been shown to be practical for real world applications [24], [9], [25], such as emergency evacuation planning.

We compared JSHOP and HATP using the Blocks World (BW), Rovers, and Towers of Hanoi (ToH) domains. The first two—standard benchmark domains from the International Planning Competition (IPC)—were provided with the JSHOP distribution, and the third was taken from [26]. We could not include Pyhop in the comparison because it did not seem able to handle non-trivial planning problems; e.g. the BW domain in the Pyhop distribution exceeded the maximum Python recursion depth when solving some 10-blocks problems. We manually translated JSHOP problems and domains into HATP encodings by following our mapping process. The three domains were sufficiently varied for the comparison: the BW domain does not require any backtrack- ing due to a domain-specific heuristic [27] that ensures a move is necessary—that once done, it will never need to be undone; the Rovers domain does require the planner to backtrack whichever choices pursued turn out to not work; and the ToH domain does not require any backtracking either, but unlike the other domains, even a 12-disk ToH problem is difficult to solve because the plan length grows exponentially with the number of disks.

Figure 5 shows our results. We believe that the HATP curve sometimes grew faster than that of JSHOP likely because of the added overhead of copying “heavier” HATP state and partial plan structures at choice points. In the ToH domain this was evident for 10 rings—the most HATP could solve with the default memory limit—where solutions needed over 1000 actions; over 500 were needed for 9 rings. Clearly, these plan lengths are sufficient for most real world applications. In the other two domains the plan lengths grew linearly with the number of waypoints/blocks. In the BW domain HATP performed similarly to JSHOP even with 500

---

9For example, when a choice point is reached during planning, the full HATP state is copied instead of just a subset of it as done in JSHOP [23], which uses heuristics such as keeping track of changes to the state.
10Experiments were done on a ThinkPad with a quad-core Intel Core i7 processor and 8 GB of RAM, running CentOS 6.5 and Java 1.7. Each data point in the figure was the average of 25 total runs from 5 random planning problems, except for the ToH domain where it was the average of 5 total runs from 1 initial state (the goal state was always the same). We calculated runtime by adding “user” and “system” times, but “system” time was negligible and excluding it made no noticeable difference to the graphs. Vertical bars indicate the (often negligible) standard deviation.
blocks, which had a domain of 500 HATP entity instances (i.e. C++ objects), one per block, and required solutions with over 1500 actions. In the Rovers domain HATP started being slower than JSHOP at about 40 waypoints, amounting to about 250 HATP entity instances, which is to our knowledge more than any integration of HATP has needed to date. At 100 waypoints, HATP problems had about 600 entity instances, which HATP was still able to solve, albeit slower than JSHOP.

VI. CONCLUSION AND FUTURE WORK
In this paper we showed how HATP offers an intuitive, and in some cases a concise syntax inspired by structured programming concepts, and also certain features to ease extensions to HATP and integration with applications written in standard programming languages. Some of the advantages highlighted include allowing relevant FOL predicates to be treated and manipulated as sets of values, allowing the members of a precondition/effect to be ordered for efficiency reasons, and an implementation that naturally realises HATP’s syntax and semantics as typical C++ data structures. Since such data structures are quite possibly “heavier” than the more specialised, list-based ones used in traditional HTN planners such as SHOP, we proceeded to show that HATP still performs acceptably, using three well known and sufficiently diverse domains, together with the JSHOP HTN planner as a baseline. Our evaluation complements the many existing extensions of HATP and integrations with real world applications, some of which were highlighted in the introduction. Finally, we showed that HATP’s conceptually distinct syntax and an important subset of its semantics is sound. To this end, we developed an unambiguous specification of the syntax and semantics and showed how they are related to the syntax and semantics of the SHOP family of planners.

We believe that HATP’s performance can be significantly improved with carefully designed C++ classes for HATP entities, and efficient copying/discarding of their corresponding C++ objects when a backtrack point is reached. Likewise, since HATP domains are compiled rather than interpreted, we can exploit certain advantages as done in JSHOP2, such as inferring from the domain how large a set (e.g. path in figure 1) could get, and compiling the set into an array rather than a C++ vector. Finally, we intend to complement our analysis of HATP’s syntax with user studies to determine how much of an improvement in programming efficiency HATP could offer over SHOP.

REFERENCES