Semantic Attachments for HTN Planning

Maurício Cecílio Magnaguagno, Felipe Meneguzzi
School of Computer Science (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre - RS, Brazil
mauricio.magnaguagno@acad.pucrs.br, felipe.meneguzzi@pucrs.br

Abstract

Hierarchical Task Networks (HTN) planning uses a decomposition process guided by domain knowledge to guide search towards a planning task. While many HTN planners allow calls to external processes (e.g. to a simulator interface) during the decomposition process, this is a computationally expensive process, so planner implementations often use such calls in an ad-hoc way using very specialized domain knowledge to limit the number of calls. Conversely, the classical planners that are capable of using external calls (often called *semantic attachments*) during planning are limited to generating a fixed number of ground operators at problem grounding time. We formalize *Semantic Attachments* for HTN planning using semi coroutines, allowing such procedurally defined predicates to link the planning process to custom unifications outside of the planner, such as numerical results from a robotics simulator. The resulting planner then uses such coroutines as part of its backtracking mechanism to search through parallel dimensions of the state-space (e.g. through numeric variables). We show empirically that our planner outperforms the state-of-the-art numeric planners in a number of domains using minimal extra domain knowledge.

Introduction

Planning in domains that require numerical variables, for example, to drive robots in the physical world, must represent and search through a space defined by real-valued functions with a potentially infinite domain, range, or both. This type of numeric planning problem poses challenges in two ways. First, the description formalisms (Fox and Long 2003) might not make it easy to express the numeric functions and its variables, resulting in a description process that is time consuming and error-prone for real-world domains (Strobel and Kirsch 2014). Second, the planners that try to solve such numeric problems must find efficient strategies to find solutions through this type of state-space. Previous work on formalisms for domains with numeric values developed the Semantic Attachment (SA) construct (Dornhege et al. 2009) in classical planning. Semantic attachments (Weyhrauch 1981) describe the attachment of an interpretation to a predicate symbol using an external procedure, allowing the planner to reason about numeric values from external functions.

Most planners are limited to purely symbolic operations, lacking structures to optimize usage of continuous resources involving numeric values (Gerevini, Saetti, and Serina 2008). Unlike discrete logical symbols, numeric values have an infinite domain and are harder to compare both conceptually and implementationally, with cumbersome *ad hoc* solutions for objects represented by several numeric values (e.g. points or polygons) that must be handled and compared as one. Planning descriptions usually simplify such complex objects to symbolic values (e.g. *p25* or *poly2*) that are easier to compare. Detailed numeric values are ignored during planning or left to be decided later (Piotrowski et al. 2016), which may force replanning (Şucan and Kavraki 2011). Instead of simplifying the description or doing multiple comparisons in the description itself, our goal is to exploit external formalisms orthogonal to the symbolic description. To achieve that we build a mapping from symbols to objects generated as we query semantic attachments. Semantic attachments are used in classical planning (Dornhege et al. 2009) to unify values analogously to predicates, and their main advantage is that domain engineers need not discern between them and common predicates. Thus, we extend classical HTN planning algorithms and their formalism to support semantic attachment queries. While external function calls map to functions defined outside the HTN description, we define SAs as semi coroutines (Dahl, Dijkstra, and Hoare 1972), subroutines that suspend and resume their state, to iterate across zero or more values provided one at a time by an external implementation, mitigating the potentially infinite range of the external function.

Our contributions are threefold. First, we introduce SAs for HTN planning to describe and efficiently evaluate external predicates at execution time. Second, we improve the readability of domains and plans and simplify processing external objects and structures using a symbol-object table. Finally, we empirically compare the resulting HTN planner with modern classical planners in a number of mixed symbolic/numeric domains showing substantial gains in speed with minimal domain knowledge.

Background

Classical planning is interested in finding sequences of transitions that transform properties of the world from an initial configuration to a goal configuration. The transitions are described by a domain expert in terms of preconditions and effects, usually using a standard language, such as PDDL (McDermott et al. 1998). Throughout this section we use the formalism from Ghallab, Nau, and Traverso (2004).

Classical planning formalisms comprise these elements:

Terms are symbols that represent objects or variables. There is a finite set of objects available.

Predicates represent relations between terms and are defined by a signature **name** applied to a sequence of N **terms**, represented by t_n, **pre** = $\langle$**name(pre), terms(t**$_1$**, ..., t**$_n$**)**$\rangle$. When all terms of a predicate are objects we call it a ground predicate and unification or grounding the process to replace variables with available objects. We call **F** the finite set of facts, comprised of all ground predicates.

State is a finite set of ground predicates that describe a world configuration at a particular time. It is represented by **S** = $\langle p_1, ..., p_n\rangle$. Partial states may be used to represent only what we are interested in, in a closed-world assumption where we have full observability, or are certain, in an open-world assumption where we may lack certainty about which predicates are true or false.

Operator represented by a 4-tuple **o** = $\langle$**name(o), pre(o), eff(o), cost(o)**$\rangle$: **name(o)** is the description or signature of **o**; **pre(o)** are the preconditions of **o**, a set of predicates that must be satisfied by the current state for action **o** to be applied; **eff(o)** are the effects of **o**; The effects contain positive and negative sets, **eff(o)+** and **eff(o)-**, that add and remove predicates from the state, respectively; Preconditions and effects may have variables as terms to generalize operators; More complex preconditions and effects consider expressions, quantifiers and conditions instead of just sets; **cost(o)** represents the cost of applying this operator, usually 1 or 0.

Action is an instantiated operator, with objects from **O** replacing variables from preconditions and effects. During the planning process, actions that have APPLICABLE preconditions in the current state can APPLY their effects to create a new possible state, deleting predicates from **eff(a)-** and adding predicates from **eff(a)+**. The finite set of actions available is called **A**.

Initial state is a complete state, in a closed-world, represented by $\mathbf{I} \subseteq \mathbf{F}$, which is defined by a set of predicates that represent the current state of the environment.

Goal state is a partial state represented by $\mathbf{G} \subseteq \mathbf{F}$, which is defined by a set of predicates that we desire to achieve by successfully applying the actions available.

Domain brings all problem independent elements together in a tuple **D** = $\langle$**F, A**$\rangle$.

Plan is the solution concept of a planning problem and is represented by a sequence of actions that when applied in an specific order will modify **I** to **G** in **D**, π = $\langle$ **a**$_1$**, ..., a**$_n$$\rangle$. An empty plan solves $\mathbf{I} \subseteq \mathbf{G}$.

Planning Instance is the triple **P** = $\langle$**D, I, G**$\rangle$: planners take as input **P** and return either π or *failure*.

Hierarchical planning shifts the focus from goal states to tasks to exploit human knowledge about problem decomposition using an hierarchy of domain knowledge recipes as part of the domain description (Nau et al. 1999). This hierarchy is composed of primitive tasks that map to operators and non-primitive tasks, which are further refined into subtasks using *methods*. The decomposition process is repeated until only primitive-tasks mapping to operators remain, which results in the plan itself, backtracking when necessary.

Unlike classical planning, hierarchical planning only considers tasks obtained from the decomposition process to solve the problem, which limits the ability to solve problems to improve execution time by evaluating a smaller number of operators. The description of an HTN planning problem is more complex than equivalent classical planning descriptions, since it includes domain knowledge with potentially recursive tasks, which allows the description of more complex problems than classical planning (Erol, Nau, and Subrahmanian 1995). HTN planning expands the elements of classical planning with:

HTN Terms are symbols that represent objects, variables or external function calls.

Task represented by a signature **name(task)** applied to a sequence of N **terms** that act as parameters, forwarding ground values to be used by the task, **task** = $\langle$**name(task), terms(t**$_1$**, ..., t**$_n$**)**$\rangle$. The task is then mapped by name to an operator (primitive task) or method (non-primitive/abstract task). A set of tasks to be decomposed by an instance is called **T**. During each step of the planning process a task is removed from **T** by SHIFT. Tasks can be removed based on a set of defined ordering constraints. In this work we limit ordering constraints to total-order, such that $t_{n-1} \prec t_n$.

Method represented by a 3-tuple **m** = $\langle$**name(m), pre(m), tasks(m)**$\rangle$: **name(m)** represents the description or signature of **m**; **pre(m)** represents the preconditions of **m**; **tasks(m)** represents the subtasks of **m**, replacing the original task for new tasks; The finite set of methods available is called **M**. During the HTN planning process, each possible DECOMPOSITION is found by searching which methods match the current task, **name(t) = name(m)** for $t \in$ SHIFT(**T**) $\wedge$ $m \in$ **M**.

External Functions are not usually included in HTN formalizations, but are found in almost all HTN planner implementations. We thus formally include external functions **E**, that allow the HTN planner to invoke external code to create new objects for the problem instance during search (e.g. to represent numbers and numeric operations). In effect, the presence of such functions makes the states in HTN planning potentially infinite, since such functions can introduce arbitrary new objects. Functions are defined by a signature **name** applied to a sequence of N **terms**, represented by t_n, **e** = $\langle$**name(e), terms(t**$_1$**, ..., t**$_n$**)** $\rangle$, and replaced by a single object obtained from their evaluation during planning.

HTN Domain also includes **M** methods and **E** external functions, is represented by $\mathbf{D} = \langle \mathbf{F, A, M, E} \rangle$.

HTN Planning Instance represented by the 3-tuple $\mathbf{P} = \langle \mathbf{D, I, T} \rangle$ and returns π or *failure*.

HTN Plan represented by a sequence of actions that when applied in an specific order will modify **I** to an implicit **G** defined by **T**, $\pi = \langle \mathbf{a}_1, ..., \mathbf{a}_n \rangle$.

Algorithm 1 corresponds to the Total-order Forward Decomposition (TFD) (Ghallab, Nau, and Traverso 2004, chapter 11). This is a recursive planner that selects one task with ordering constraints satisfied and either updates the state for primitive tasks or decomposes non-primitive tasks.

Algorithm 1 Total-order Forward Decomposition planner

```
1:  function TFD(S, T, D)
2:    if T = ∅ then return empty $\pi$
3:    t ← SHIFT(T)
4:    if t is a primitive task
5:      for $t_{applicable}$ ∈ APPLICABLE(pre(t), S) do
6:        $\pi$ ← TFD(APPLY($t_{applicable}$, S), T, D)
7:        if $\pi \neq$ failure then return $t_{applicable} \cdot \pi$
8:    else if t is a non-primitive task
9:      for m ∈ DECOMPOSITION(t, D) do
10:       for tasks(m) ∈ APPLICABLE(pre(m), S) do
11:         $\pi$ ← TFD(S, tasks(m) · T, D)
12:         if $\pi \neq$ failure then return $\pi$
13:   return failure
```

Coroutines

Coroutine is a cooperative multitasking approach that gives the programmer full execution control, as routines are not preempted by a scheduler but called from other routines, with persistent state between calls (Moura and Ierusalimschy 2009). Generators are limited coroutines that always pass control back to the caller while yielding a value, and thus are usually implement iterators. Every time the generator executes a *yield* instruction it saves the state to the stack, suspends, and resumes in the next call. The generator finishes once either the generator or the caller returns.

Unification functions, such as APPLICABLE, can be implemented as a generator that yield ground instances of an operator or method with preconditions satisfied by the current state. HTN backtracking causes a resume operation in the generator to try other unifications or returning, once no more unifications are found in the current level of recursion.

Symbolic-Geometric Planning

Classical planners with heuristic functions can solve mixed symbolic-numeric problems efficiently using a process of discretization. A discretization process converts continuous values into sets of discrete symbols at often predefined granularity levels that vary between different domains. However, if the discretization process is not possible, one must use a planner that also supports numeric features, which requires another heuristic function, description language and usually more computing power due to the number of states generated by numeric features (Hoffmann 2003). Numeric features are especially important in domains where one cannot discretize the representation and usually appear in geometric or physics subproblems of a domain and cannot be avoided during planning. Unlike symbolic approaches where literals are compared for equality during precondition evaluation, numeric value comparison is non-trivial. To avoid doing such comparison for every numeric value, the user is left responsible for explicitly defining when one must consider rounding errors, which impacts description time and complexity. For complex object instances (in the object-oriented programming sense), such as polygons that are made of point instances, comparison details in the description are error-prone. Details such as the order of polygon points and floating point errors in their coordinates are usually irrelevant for the planner and the domain designer, and should not be part of the symbolic domain description as they are part of a non-symbolic specification.

Semantic Attachments

The non-symbolic specifications required for symbolic-geometric planning can be implemented by external function calls to improve what can be expressed and computed by an HTN planner. Such functions come with disadvantages, as they are not expected to keep their own state, returning a single value solely based on the provided parameters. While HTN planners can abstract away the numeric details via external function calls, there are limitations to this approach if a particular function is used in a decomposition tree where it is expected to backtrack and try new values from the function call (i.e. if the function is meant to be used to generate multiple terms as part of the search strategy). An external function must return a list of values to account for all possible decompositions so the planner tries one at a time until either one value succeeds or all values lead to failure. Generating a complete list for such external calls creates two key problems. First, this is too costly when compared to computing a single value, as the first value could be enough to find a feasible plan. Second, the HTN methods must handle such lists of values explicitly, introducing unnecessary and undesirable complexity into the domain engineering process. A Semantic Attachment, on the other hand, acts as an external predicate that unifies with one possible set of values at a time, rather than storing a complete list of possible sets of values in the state structure. This implementation saves time and memory during planning, as only backtracking causes the external coroutine to resume generating new unifications until a certain amount of plans are found. Each SA acts as a black box that simulates part of the environment encoding the results in state variables that are often orthogonal to other predicates (Francès et al. 2017). While common predicates are stored in a state structure, SAs are computed at execution time by coroutines. Using SAs allows us to encode states that contain symbolic information readily understandable for humans, as well as abstract away complex state-based operations to external methods. The set of coroutines available at planning time that verify ground terms or unify each set of possible values for free variables, one at a time, is represented by **SA**. Each SA is defined by a signature **name** applied to a sequence of N **terms**, represented by t_n, **sa** = $\langle$**name(sa), terms(t$_1$, ..., t$_n$)**$\rangle$. In summary, external functions do not naturally work as semantic attachments, since their return value is expected to be a single unique value for multiple calls with the same parameters. The implicit assumption for most external function calls is that they are stateless, so in order to return multiple values one must design the function to return a list with all the possible values and to describe the logic (in the planning domain) that will consume/use each of these values from the list. Semantic Attachments on the other hand work as externally defined predicates that can unify one set of free variables at a time or test a ground predicate externally. This different approach removes the burden of computing all values, computing only until HTN decomposition can progress, and not having to deal a list of values, that encumbering the HTN domain description unrelated mechanisms to the domain, such as list operations and sorting. The domain designer can define its preferences directly in the semantic attachment, to try certain values first, instead of generating multiple values and sorting them before planning resumes. Thus we can rely on the backtracking semantics of the HTN planning procedure to iterate over possible values of the semantic attachment, resulting in a cleaner and more intuitive domain description.

Symbolic anchors for external values

We abstract away the numeric parts of the planning process encoded through SAs in a layer between the symbolic planner and external libraries. We leverage the three-layered architecture of Figure 1 inspired by the work of de Silva and Meneguzzi (2015). In the symbolic layer we manipulate an anchor symbol as a term, such as *polygon1*, while in the external layer we manipulate the corresponding *Polygon instance with N points* as a geometric object based on what the selected external library specifies. With this approach we avoid complex representations in the symbolic layer and the extra overhead in the domain description. Instances created by the external layer that must be exposed to the symbolic layer are compared with stored object instances to reuse a previously defined symbol or create a new one, i.e. always represent position $\langle 2, 5\rangle$ as *p1*. This process makes symbol comparison work in the symbolic layer even for symbols related to complex external objects. The symbol-object table is responsible for storing the anchors between symbols and object instances to be used by external function calls and SAs. The planner maintains a global and consistent view of this table during the planning process, as each unique symbol will map the same internal object, even if such symbol is discarded in one decomposition branch. Our symbol-object table is defined by N key-value pairs, represented by **SO-table** = $\langle$$\langle$**key(k$_1$),value(v$_1$)**$\rangle$, ..., $\langle$**key(k$_n$),value(v$_n$)**$\rangle$$\rangle$, with QUERY and INSERT functions to find pairs based on anchor symbols as keys and to add anchors between generated symbols and external objects, respectively. Once operations are finished in the external layer the process happens in reverse order, objects are transformed back into symbols that are exposed by free variables. The intermediate layer acts as the foreign function interface between the two layers,

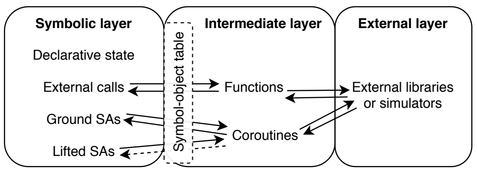

Figure 1: Symbolic and external layers share information through an intermediate layer that maps representations and calls between them.

and can be modified to accommodate other external libraries without modifications to the symbolic description.

The symbolic description comes from domain and problem files using UJSHOP, our extended JSHOP (Ilghami and Nau 2003) input language with SA support. The symbolic description is compiled before planning. The compilation step is taken by HYPE, a framework that parses the description to an intermediate representation and compile[1] to an equivalent Ruby code that connects with our planner, an HTN planner with the features described later in this section. The parser and compiler can be replaced for equivalent modules to eventually support other input or output formats. External functions, SAs and the SO-table are defined in a separate file outside the compilation process, only loaded during planning, making **D** = $\langle \mathbf{F}, \mathbf{A}, \mathbf{M}, \mathbf{E}, \mathbf{SAs}, \mathbf{SO\text{-}table}\rangle$. The process and its elements are better seen in Figure 2 illustrates the HYPERTENSIONN_U framework.

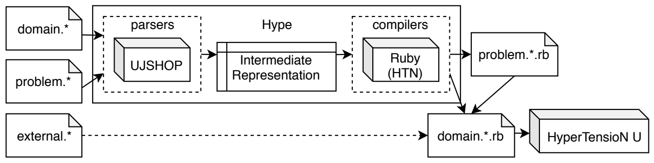

Figure 2: Framework parses symbolic description and compiles to the target language before connecting with external definitions and HTN planner.

Reordering preconditions

SAs can work as interpreted predicates (Mohr et al. 2018), evaluating the truth value of a predicate procedurally, and also grounding free variables. If we restrict our operators to the STRIPS-style ones, we can reorder the preconditions during the compilation phase to improve execution time, removing the burden of the domain designer to optimize a mostly declarative description by hand, based on how free variables are used as terms. Each free variable creates a dependency between the first predicate or SA that contains such variable as a term and the next predicates or SAs that contain the same term. The first predicate or SA is responsible for grounding such variable while the next predicates or SAs only verify if the previously ground value matches with the current state. Predicates have priority over SAs to ground

[1]The compiler here is a source-to-source compiler or *transpiler*, as code is only translated from one language to another.

Algorithm 2 Filter preconditions during compilation phase based on free variables used as terms of predicates and SAs.

```
1: function FILTERPRECONDITIONS(pre)
2:     $P_{ground} \leftarrow \{p \mid p \in pre \wedge (p \in \mathbf{F} \vee (p \in \mathbf{E} \wedge$
3:         $\forall t \in \mathbf{terms}(p)\ t = object))\}$
4:     $P_{lifted} \leftarrow \{p \mid p \in pre \wedge (p \in Predicate \vee p \in \mathbf{E}) \wedge$
5:         $\exists t \in \mathbf{terms}(p)\ t \neq object\}$
6:     $P_{sa} \leftarrow \{p \mid p \in pre \wedge p \in \mathbf{SA}\}$
7:     return $\langle P_{ground}, P_{lifted}, P_{sa} \rangle$
```

Listing 1: Abstract method with SAs among preconditions.

```
(:attachments (sa1 ?a ?b) (sa2 ?a ?b))
(:method (m ?t1 ?t2)
  label
  (; preconditions
    (call != ?t1 ?t2) ; no dependencies
    (call != ?fv1 ?fv2) ; ?fv1 and ?fv2 dependencies
    (sa1 ?t1 ?fv1)    ; no dependencies, ground ?fv1
    (pre1 ?t1 ?t2)    ; no dependencies
    (sa2 ?fv1 ?fv2)   ; ?fv1 dependency, ground ?fv2
    (pre2 ?fv3 ?fv1)) ; ?fv1 dependency, ground ?fv3
  (; subtasks
    (subtask ?t1 ?t2 ?fv1 ?fv2)))
```

free variables, as the possible values are obtained from the current (finite) state, while SAs may cover a possibly infinite number of values. Algorithm 2 shows how preconditions are filtered in distinct sets used to reorder them for performance.

Consider the abstract method example of Listing 1, with two SAs among preconditions, *sa1* and *sa2*. The compiled output shown in Algorithm 3 has both SAs evaluated after common predicates, while function calls happen before or after each SA, based on which variables are ground at that point. In Line 4 the free variables *fv1* and *fv3* have a ground value that can only be read and not modified by other predicates or SAs. In Line 7 every variable is ground and the second function call can be evaluated.

Each SA is responsible for unifying all remaining free variables with valid values before resuming and have a stop condition, otherwise the HTN process will keep backtracking and evaluating the SA seeking new values and never returning failure. Due to the implementation support of arbitrary-precision arithmetic and accessing data from real-world streams of data/events (which are always new and potentially infinite) a valid value may never be found, and we expect the domain designer to implement mechanisms to limit the maximum number of times a SA might try to evaluate a call (i.e. to have finite stop conditions). This maximum number of tries can be implemented as a counter in the internal state of an SA to avoid repeated evaluation of the same values. Note that the number of side-effects in both external functions calls and SAs increases the complexity of correctness proofs and the ability to inspect and debug domain descriptions.

Algorithm 3 Compiled preconditions from Listing 1, reordered to optimize execution time.

```
1: function M($S, t1, t2$)
2:     if t1 $\neq$ t2
3:         for each fv1, fv3; $\{\langle$pre1,t1,t2$\rangle$,$\langle$pre2,fv3,fv1$\rangle\} \subseteq S$ do
4:             for each sa1(t1, fv1) do
5:                 free variable fv2
6:                 for each sa2(fv1, fv2) do
7:                     if fv1 $\neq$ fv2 then yield [$\langle$subtask, t1, t2, fv1, fv2$\rangle$]
```

Algorithm 4 SA-enabled APPLICABLE tests ground, lifted and coroutine based preconditions to replace free variables by suitable values.

```
1: function APPLICABLE(pre, S, SO-table)
2:     $P_{ground}, P_{lifted}, P_{sa} \leftarrow$ FILTERPRECONDITIONS(pre)
3:     if $P_{ground} \nsubseteq$ S then return
4:     for $pre_{lifted}$ with free variables $fvs_{lifted} \in P_{lifted}$ do
5:         for each $fvs_{lifted}$ match with name($pre_{lifted}$) $\in$ S do
6:             for $pre_{sa}$ with free variables $fvs_{sas} \in P_{sa}$ do
7:                 for EXTERNAL($pre_{sa}$, $fvs_{sas}$, S, SO-table) do
8:                     yield
```

Unlike Algorithm 1 we move the generic APPLICABLE from the HTN algorithm to custom unifiers implemented by the compilation step directly into the operator and method functions, with preconditions reordered based on the return of Algorithm 2. This is an important modification to allow more complex preconditions to be evaluated. By moving such routine to the method itself we can have custom implementations, including generator-based implementations. The original Algorithm 1 does not define how the APPLICABLE set of free variable assignment can be implemented. The equivalent non-custom version of our approach is defined by Algorithm 4. The symbol-object table is an argument of the new APPLICABLE routine, to be used by external function calls and SAs after ground preconditions are satisfied and lifted preconditions used to evaluate some of the free variables present in the preconditions.

Examples

Discrete distance between objects

A common problem when moving in dynamic and continuous environments is to check for object collisions, as agents and objects do not move across tiles in a grid. One solution is to calculate the distance between the centroid of both objects and verify if this value is in a safe margin before considering which action to take. To avoid the many geometric elements involved in this process we can map centroid position symbols to coordinate instances and only check the symbol returned from the symbol-object table, ignoring specific numeric details and comparing a symbol to verify if objects are near enough to collide. This process is illustrated in Figure 3, in which p_0 and p_1 are centroid position symbols that match symbols S_0 and S_1 in the symbol-object table, which maps their value to point objects O_0 and O_1. Such internal objects are used to compute *distance* and return a symbolic distance in situations where the actual numeric value is unnecessary.

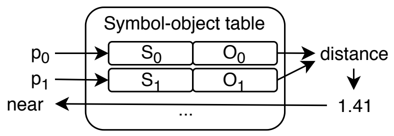

Figure 3: The symbol to object table maps symbols to object-oriented programming instances to hide procedural logic from the symbolic layer.

An iterator for HTN

In order to find a correct number to match a spatial or temporal constraint one may want to describe the relevant interval and precision to limit the amount of possibilities without having to discretely add each value to the state. Planning descriptions usually do not contain information about numeric intervals and precision, and if there is a way to add such information it is through the planner itself, as global definitions applied to all numeric functions, i.e. timestep, mantissa and exponent digits of DiNo (Piotrowski et al. 2016). The STEP SA described in Algorithm 5 addresses this problem, unifying t with one number at a time inside the given interval with an ϵ step.

Algorithm 5 The STEP SA replaces the pointer of t with a numeric symbol before resuming control to the HTN.

```
1: function STEP($t, min = 0, max = \infty, \epsilon = 1$)
2:     for i ← $min$ to $max$ step $\epsilon$ do
3:         $t$ ← INSERT(SO-table, i)
4:         yield                                  ▷ Resume HTN
```

Lazy adjacency evaluation

To avoid having complex effects in the move operators one must not update adjacencies between planning objects during the planning process. Instead one must update only the object position, deleting the old position and adding the new position. Such positions come from a partitioned space, previously defined by the user. The positions and their adjacencies are either used to generate and store ground operators or stored as part of the state. To avoid both one could implement adjacency as a coroutine while hiding numeric properties of objects, such as position. Algorithm 6 shows the main two cases from planning descriptions. In the first case both symbols are ground, and the coroutine resumes when both objects are adjacent, doing nothing otherwise, failing the precondition. In the second case $s2$, the second symbol, is free to be unified using $s1$, the first symbol, and a set of directions D to yield new positions to replace $s2$ pointer with a valid position, one at a time. Thus, this coroutine either checks whether $s2$ is adjacent to $s1$ or tries to find a value adjacent to $s1$ binding it to $s2$ if such value exists.

Domains and Experiments

We conducted empirical tests with our own HTN planner[2] in a machine with Dual 6-core Xeon CPUs @2GHz / 48GB

[2]github.com/Maumagnaguagno/HyperTensioN_U.

Algorithm 6 This ADJACENT SA implementation can either check if two symbols map to adjacent positions or generate new positions and their symbols to unify $s2$.

```
1: D ← {(-1,-1),(0,-1),(1,-1),(-1,0),(1,0),(-1,1),(0,1),(1,1)}
2: function ADJACENT($s1$, $s2$)
3:     $s1$ ← QUERY(SO-table, $s1$)
4:     if $s2$ is ground
5:         $s2$ ← QUERY(SO-table, $s2$)
6:         if $|x(s1) - x(s2)| \leq 1 \land |y(s1) - y(s2)| \leq 1$ then yield
7:     else if $s2$ is free
8:         for each (x, y) $\in$ D do
9:             nx ← x + x($s1$);    ny ← y + y($s1$)
10:            if $0 \leq nx < WIDTH \land 0 \leq ny < HEIGHT$
11:                $s2$ ← INSERT(SO-table, ⟨nx, ny⟩)
12:                yield
```

memory, repeating experiments three times to obtain an average. The results show a substantial speedup over the original classical description from ENHSP (Scala et al. 2016) with more complex descriptions, while being competitive against Metric-FF (Hoffmann 2003).

Plant Watering / Gardening

In the Plant Watering domain (Frances and Geffner 2015) one or more agents move in a 2D grid-based scenario to reach taps to obtain certain amounts of water and pour water in plants spread across the grid. Each agent can carry up to a certain amount of water and each plant requires a certain amount of water to be poured. Many state variables can be represented as numeric fluents, such as the coordinates of each agent, tap and plant, the amount of water to be poured and being carried by each agent, and the limits of how much water can be carried and the size of the grid. There are two common problems in this scenario, the first is to travel to either a tap or a plant, the second is the top level strategy. To avoid considering multiple paths in the decomposition process one can try to move straight to the goal first, and only to the goal in scenarios without obstacles, which simplifies the travel method. To achieve this straightforward movement we modify the ADJACENT SA to consider the goal position also, using an implementation of Algorithm 7. The top level strategy may consider which plant is closer to a tap or closer to an agent, how much water an agent can carry and so on. The simpler top level strategy is to verify how much water must be poured to a plant, travel to a tap, load water, travel to the previously selected plant and pour all the water loaded. Repeating this process until every plant has enough water poured. The travel method is shown in Listing 2 and compiled to Algorithm 8. We compare with the fastest satisficing configuration of ENHSP (`sat`) and Metric-FF 2.1 (`standard-FF`) in Figure 4, which shows that our approach is faster with execution times near 0.01s (ignoring interpreter loading time), or competitive with Metric-FF around 0.11s (considering interpreter loading time), with all three planners obtaining non-step-optimal plans.

Algorithm 7 In this goal-driven ADJACENT SA the positions are coordinate pairs, and two variables must be unified to a closer to the goal position in an obstacle-free scenario.

```
function ADJACENT($x, y, nx, ny, gx, gy$)
    $x \leftarrow$ QUERY(**SO-table**, $x$); $y \leftarrow$ QUERY(**SO-table**, $y$)
    $gx \leftarrow$ QUERY(**SO-table**, $gx$); $gy \leftarrow$ QUERY(**SO-table**, $gy$)
    $\triangleright$ COMPARE returns -1, 0, 1 for $<, =, >$, respectively
    $nx \leftarrow$ INSERT(**SO-table**, $x$ + COMPARE($gx$, $x$))
    $ny \leftarrow$ INSERT(**SO-table**, $y$ + COMPARE($gy$, $y$))
    yield
```

Listing 2: Excerpt of the Plant Watering HTN domain, the ADJACENT SA is described separately.

```
(:attachments (adjacent ?x ?y ?nx ?ny ?gx ?gy))
(:method (travel ?a ?gx ?gy)
  base
  (; preconditions
    (call = (call function (x ?a)) ?gx)
    (call = (call function (y ?a)) ?gy) )
  () ; empty subtasks
  keep_moving
  (; preconditions
    (adjacent (call function (x ?a))
      (call function (y ?a)) ?nx ?ny ?gx ?gy))
  (; subtasks
    (!move ?a ?nx ?ny)
    (travel ?a ?gx ?gy)))
```

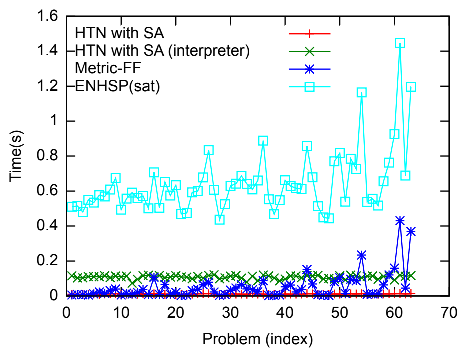

Figure 4: Time in seconds to solve Plant Watering problems.

Car Linear

In the Car Linear domain (Bryce et al. 2015) the goal is to control the acceleration of a car, which has a minimum and maximum speed, without external forces applied, only moving through one axis to reach its destination, and requiring a small speed to safely stop. The idea is to propagate process effects to state functions, in this case acceleration to speed and speed to position, while being constrained to an acceptable speed and acceleration. The planner must decide when and for how long to increase or decrease acceleration, therefore becoming a temporal planning problem. We use a STEP SA to iterate over the time variable and propagate temporal effects and constraints, i.e. speed at time t. We compare the execution time of our approach with ENHSP with `aibr`, ENHSP main configuration for planning with autonomous processes, in Table 1. There is no comparison with a native HTN approach, as one would have to add a discrete finite set of time predicates (e.g. ⟨*time 0*⟩) to the initial state description to be selected as time points during planning.

Algorithm 8 Compiled output of the Plant Watering HTN domain excerpt from Listing 2.

```
function TRAVEL($a, gx, gy$)
    if x($a$) = $gx$ ∧ y($a$) = $gy$ then yield ∅
    else
        free variables nx, ny
        for each ADJACENT(x($a$), y($a$), nx, ny, $gx$, $gy$) do
            yield [⟨move, $a$, nx, ny⟩, ⟨travel, $a$, $gx$, $gy$⟩]
```

Problem	1	2	3	4	5	6	7	8	9
ENHSP (aibr)	0.484	0.432	0.411	0.443	0.461	0.474	0.465	0.436	63.585
HTN with SA	0.016	0.019	0.014	0.016	0.018	0.019	0.017	0.018	01.402

Table 1: Time in seconds to solve Car Linear problems.

Conclusion

We developed a notion of semantic attachments for HTN planners that not only allows a domain expert to easily define external numerical functions for real-world domains, but also provides substantial improvements on planning speed over comparable classical planning approaches. The use of semantic attachments improves the planning speed as one can express a potentially infinite state representation with procedures that can be exploited by a strategy described as HTN tasks. As only semantic attachments present in the path decomposed during planning are evaluated, a smaller amount of time is required when compared with approaches that precompute every possible value during operator grounding. Our description language is arguably more readable than the commonly used strategy of developing a domain specific planner with customized heuristics, or attaching procedures that must have all variables ground (Ghallab, Nau, and Traverso 2004, chapter 11). Specifically, we allow designers to easily define external functions in a way that is readable within the domain knowledge encoded in HTN methods at design time, and also dynamically generate symbolic representations of external values at planning time, which makes generated plans easier to understand.

Our work is the first attempt at defining the syntax and operation of semantic attachments for HTNs, allowing further research on search in SA-enabled domains within HTN planners. This kind of extension comes in line with recent attempts at including more expressive planning languages than vanilla PDDL, and which include more functional elements in it (e.g. Tarski[3]). Dornhege et al. (2009) uses semantic attachments to compute the truth value of propositions in preconditions, and effects on numerical fluents that would be too complex to describe otherwise. Dornhege, Hertle, and Nebel (2013) adds action grounding and action costs as semantic attachment modules. By contrast, our method can be

[3]github.com/aig-upf/tarski

used to both compute the truth value of a ground proposition or to unify free-variables with values in preconditions. Such unified values can be symbolic or numeric, and can be generated indefinitely, which is useful when searching for a numeric value that satisfies some property as one can define the equivalent of an iterator as a predicate. Further the grounding action approach (Dornhege, Hertle, and Nebel 2013) does not match the HTN domain style, where most operator parameters (groundings) are decided by method preconditions before decomposing to primitive tasks.

Future work includes implementing a cache to reuse previous values from external procedures applied to similar previous states (Dornhege, Hertle, and Nebel 2013) and a generic construction to access such values in the symbolic layer, to obtain data from explored branches outside the state structure, i.e. to hold mutually exclusive predicate information. We plan to develop more domains, with varying levels of domain knowledge and SA usage, to obtain better comparison with other planners and their resulting plan quality. The advantage of being able to exploit external implementations conflicts with the ability to incorporate such domain knowledge into heuristic functions, as such knowledge is outside the description. Further work is required to expose possible metrics from a SA to heuristic functions.

Acknowledgements: We acknowledge the support given by CAPES/Pro-Alertas (88887.115590/2015-01) and CNPQ within process number 305969/2016-1 under the PQ fellowship. This paper was achieved in cooperation with HP Brasil Indústria e Comércio de Equipamentos Eletrônicos LTDA. using incentives of Brazilian Informatics Law (Law n^{o} 8.2.48 of 1991).

References

Bryce, D.; Gao, S.; Musliner, D. J.; and Goldman, R. P. 2015. SMT-Based Nonlinear PDDL+ Planning. In *AAAI*, 3247–3253.

Dahl, O.-J.; Dijkstra, E. W.; and Hoare, C. A. R. 1972. *Structured programming*. Academic Press Ltd.

de Silva, L., and Meneguzzi, F. 2015. On the design of symbolic-geometric online planning systems. In *2015 Workshop on Hybrid Reasoning (HR 2015)*, 1–8.

Dornhege, C.; Gissler, M.; Teschner, M.; and Nebel, B. 2009. Integrating symbolic and geometric planning for mobile manipulation. In *Safety, Security & Rescue Robotics (SSRR), 2009 IEEE International Workshop on*, 1–6. IEEE.

Dornhege, C.; Hertle, A.; and Nebel, B. 2013. Lazy evaluation and subsumption caching for search-based integrated task and motion planning. In *IROS workshop on AI-based robotics*.

Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence* 76(1-2):75–88.

Fox, M., and Long, D. 2003. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*.

Frances, G., and Geffner, H. 2015. Modeling and computation in planning: Better heuristics from more expressive languages. In *ICAPS*, 70–78.

Francès, G.; Ramírez, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely declarative action descriptions are overrated: Classical planning with simulators. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 4294–4301.

Gerevini, A. E.; Saetti, A.; and Serina, I. 2008. An approach to efficient planning with numerical fluents and multi-criteria plan quality. *Artificial Intelligence* 172(8-9):899–944.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated planning: theory & practice*. Elsevier.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research (JAIR)* 20:291–341.

Ilghami, O., and Nau, D. S. 2003. A general approach to synthesize problem-specific planners. Technical report, DTIC Document.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL-the planning domain definition language. Technical report, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Mohr, F.; Lettmann, T.; Hüllermeier, E.; and Wever, M. 2018. Programmatic Task Network Planning. In *Proceedings of the 1st ICAPS Workshop on Hierarchical Planning*, 31–39.

Moura, A. L. D., and Ierusalimschy, R. 2009. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31(2):6.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial Intelligence-Volume 2*, 968–973. Morgan Kaufmann Publishers Inc.

Piotrowski, W. M.; Fox, M.; Long, D.; Magazzeni, D.; and Mercorio, F. 2016. Heuristic Planning for PDDL+ Domains. In *AAAI Workshop: Planning for Hybrid Systems*.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramirez, M. 2016. Interval-based relaxation for general numeric planning. In *ECAI*, 655–663.

Strobel, V., and Kirsch, A. 2014. Planning in the wild: modeling tools for PDDL. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, 273–284. Springer.

Şucan, I. A., and Kavraki, L. E. 2011. Mobile manipulation: Encoding motion planning options using task motion multigraphs. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 5492–5498. IEEE.

Weyhrauch, R. W. 1981. Prolegomena to a theory of mechanized formal reasoning. In *Readings in Artificial Intelligence*. Elsevier. 173–191.