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 Serena 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 (Sucan 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.
Plan is the solution concept of a planning problem and is brought all problem independent elements together. Domain is a complete state, in a closed-world, representation of the current state of the environment. Initial state is a finite set of ground predicates that describe a state that act as parameters, forwarding ground values to be used by the task. Operator represented by a 4-tuple $o = (\text{name}(o), \text{pre}(o), \text{eff}(o), \text{cost}(o))$: $\text{name}(o)$ is the description or signature of $o$; $\text{pre}(o)$ are the preconditions of $o$, a set of predicates that must be satisfied by the current state for action $o$ to be applied; $\text{eff}(o)$ are the effects of $o$. The effects contain positive and negative sets, $\text{eff}(o)^+$ and $\text{eff}(o)^-$, 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; $\text{cost}(o)$ represents the cost of applying this operator, usually 0 or 1.

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 $\text{eff}(a)^-$ and adding predicates from $\text{eff}(a)^+$. The finite set of actions available is called $A$.

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

Goal state is a partial state represented by $G \subseteq F$, 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 $D = (F, A)$.

Plan is the solution concept of a planning problem and is represented by a sequence of actions that when applied in a specific order will modify $I$ to $G$. $\pi = (a_1, ..., a_n)$. An empty plan solves $I \subseteq G$.

Planning Instance is the triple $P = (D, I, G)$: planners take as input $P$ and return either $\pi$ 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 $\text{name}(\text{task})$ applied to a sequence of $N$ terms that act as parameters, forwarding ground values to be used by the task. $\text{task} = (\text{name}(\text{task}), \text{terms}(t_1, ..., t_n))$. 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 $\text{shift}$. 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 $m = (\text{name}(m), \text{pre}(m), \text{tasks}(m))$: $\text{name}(m)$ represents the description or signature of $m$; $\text{pre}(m)$ represents the preconditions of $m$; $\text{tasks}(m)$ 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 $\text{decomposition}$ is found by searching which methods match the current task, $\text{name}(t) = \text{name}(m)$ for $t \in \text{shift}(T) \land m \in M$.

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 $\text{name}(\text{function})$ applied to a sequence of $N$ terms, represented by $t_n$, $f = (\text{name}(f), \text{terms}(t_1, ..., t_n))$, 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 D = (F, A, M, E).

HTN Planning Instance represented by the 3-tuple P = (D, I, T) 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, π = (a1, ..., an).

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 π
3:   t ← SHIFT(T)
4:   if t is a primitive task
5:     for tapplicable ∈ APPLICABLE(pre(t), S) do
6:       π ← TFD(APPLY(tapplicable, S), T, D)
7:     if π = failure then return tapplicable · π
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:           π ← TFD(S, tasks(m) · T, D)
12:       if π = failure then return π
13:   return failure

Coroutines
Coroutines 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 Jeroslaw-sch 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 \textbf{name} applied to a sequence of N \textbf{terms}, represented by \( t_1, \ldots, t_n \). 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 \textit{polygon1}, while in the external layer we manipulate the corresponding \textit{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 \((2,5)\) as \(p_1\). 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 \textit{SO-table} = \((\text{key}(k_1),\text{value}(v_1)), \ldots, (\text{key}(k_n),\text{value}(v_n))\), with \textit{query} and \textit{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, 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 \(D = \{F, A, M, E, SAs, SO-table\}\). The process and its elements are better seen in Figure 2 illustrates the HyperTension_U framework.

**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_{\text{ground}} \leftarrow \{ p | p \in \text{pre} \land (p \in F \lor (p \in E \land \forall t \in \text{terms}(p) (t = \text{object})) \}
3: \forall t \in \text{terms}(p) (t \neq \text{object})
4: \exists t \in \text{terms}(p) (t \neq \text{object})
5: P_{\text{lifted}} \leftarrow \{ p | p \in \text{pre} \land (p \in \text{Predicate} \lor p \in E) \land \}
6: \exists t \in \text{terms}(p) (t \neq \text{object})
7: P_{\text{sa}} \leftarrow \{ p | p \in \text{pre} \land \exists p \in \text{SA} \}
8: return (P_{\text{ground}}, P_{\text{lifted}}, P_{\text{sa}})

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 \( fvl, fv3; \{ \langle \text{pre1}, t1, t2 \rangle, \langle \text{pre2}, t3, \text{fv1} \rangle \} \subseteq S \) do
4: for each \( \text{sa1}(t1, t2) \) do
5: free variable \( fvl \)
6: for each \( \text{sa2}(fvl, fv2) \) do
7: if \( fvl \neq fv2 \) then yield \{ \langle \text{subtask}, t1, t2, fvl, fv2 \rangle \}

Listing 1: Abstract method with SAs among preconditions.

```
{\text{attachments} \{ \text{sa1} ?a ?b \} \{ \text{sa2} ?a ?b \}}
{\text{method} \{ \text{m} ?t1 ?t2 \}}
{\text{label} \{ ; \text{preconditions} \}}
{\text{call} \langle ! ?t1 ?t2 \rangle ; \text{no dependencies} \}
{\text{call} \langle ! ?fvl ?fv2 \rangle ; ?fvl and ?fv2 dependencies \}
{\text{sa1} ?t1 ?fvl \} ; \text{no dependencies, ground ?fvl} \}
{\text{sa2} ?fvl ?fv2} ; \text{fvl dependency, ground ?fv2} \}
{\text{pre2} ?fv3 \} ; \text{fvl dependency, ground ?fv3} \}
{; \text{subtasks} \}
{\text{subtask} \langle t1 ?t2 ?fvl ?fv2 \rangle \}}
```

Example

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: \( P_{\text{ground}}, P_{\text{lifted}}, P_{\text{sa}} \leftarrow \text{FILTERPRECONDITIONS(pre)} \)
3: if \( P_{\text{ground}} \subseteq S \) then return
4: for \( \text{pre} \in \text{lifted} \) with free variables \( \text{fvs} \in P_{\text{lifted}} \) do
5: for each \( \text{fvs} \) match with \( \text{name} \) \( \text{pre} \) \( \in S \) do
6: \( \text{for \text{pre}_{sa}} \) with free variables \( \text{fvs}_{sa} \in P_{\text{sa}} \) do
7: \( \text{for \text{EXTERNAL}} \left( \text{pre}_{sa}, \text{fvs}_{sa}, S, \text{SO-table} \right) \) do
8: yield

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.
Domains and Experiments

We conducted empirical tests with our own HTN planner\footnote{github.com/Maumagnaguagno/HyperTensioN\_U} in a machine with Dual 6-core Xeon CPUs @2GHz / 48GB.

Algorithm 5 The \textsc{step} SA replaces the pointer of \( t \) with a numeric symbol before resuming control to the HTN.

1: \textbf{function} \textsc{step} (\( t, \min = 0, \max = \infty, \epsilon = 1 \))
2: \textbf{for} \( i \leftarrow \min \) \textbf{to} \( \max \) \textbf{step} \( \epsilon \) \textbf{do}
3: \( t \leftarrow \text{\textsc{insert}}(\text{SO-table}, i) \)
4: \textbf{yield} \hfill \text{\( t \) 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 \( s_2 \), the second symbol, is free to be unified using \( s_1 \), the first symbol, and a set of directions \( D \) to yield new positions to replace \( s_2 \) pointer with a valid position, one at a time. Thus, this coroutine either checks whether \( s_2 \) is adjacent to \( s_1 \) or tries to find a value adjacent to \( s_1 \) binding it to \( s_2 \) if such value exists.

Algorithm 6 This \textsc{adjacent} SA implementation can either check if two symbols map to adjacent positions or generate new positions and their symbols to unify \( s_2 \).

1: \( D \leftarrow \{(-1,-1),(0,-1),(-1,0),(1,0),(-1,1),(0,1),(1,1)\} \)
2: \textbf{function} \textsc{adjacent}(\( s_1, s_2 \))
3: \( s_1 \leftarrow \text{\textsc{query}}(\text{SO-table, } s_1) \)
4: \textbf{if} \( s_2 \) is \textbf{ground}
5: \( s_2 \leftarrow \text{\textsc{query}}(\text{SO-table, } s_2) \)
6: \textbf{if} \( |x(s_1) - x(s_2)| \leq 1 \land |y(s_1) - y(s_2)| \leq 1 \) \textbf{then yield}
7: \textbf{else if} \( s_2 \) is \textbf{free}
8: \textbf{for} each \((x, y) \in D\) \textbf{do}
9: \( nx \leftarrow x + x(s_1); \ ny \leftarrow y + y(s_1) \)
10: \textbf{if} \( 0 \leq nx < \text{WIDTH} \land 0 \leq ny < \text{HEIGHT} \)
11: \( s_2 \leftarrow \text{\textsc{insert}}(\text{SO-table, } (nx, ny)) \)
12: \textbf{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 \textsc{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 satisfying configuration of ENHSP (\textsc{sat}) and Metric-FF 2.1 (standard-\textsc{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.
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.

Algorithm 7

1: function adjacent(x, y, nx, ny, gx, gy)
2: x ← QUERY(SO-table, x); y ← QUERY(SO-table, y)
3: gx ← QUERY(SO-table, gx); gy ← QUERY(SO-table, gy)
4: if COMPARE returns -1, 0, 1 for $<=$, $>$, respectively
5: $nx ← INSERT(SO-table, x + COMPARE(gx, x))$
6: $ny ← INSERT(SO-table, y + COMPARE(gy, y))$
7: 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
(calls - (call function (x ?a)) ?gx)
(calls - (call function (y ?a)) ?gy)
() ; empty subtasks
keep_moving
{; preconditions
(adjacent (call function (x ?a))
{; 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. $\langle \text{time} \ 0 \rangle$) to the initial state description to be selected as time points during planning.

Algorithm 8

```
1: function TRAVEL(a, gx, gy)
2: if x(a) = gx ∧ y(a) = gy then yield ⊥
3: else
4: free variables nx, ny
5: for each adjacent(x(a), y(a), nx, ny, gx, gy) do
6: yield [(move, a, nx, ny), (travel, a, gx, gy)]
```

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’s). 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 using incentives of Brazilian Informatics Law (Law 8.2.48 of 1991).

References
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.