GTPyhop: A Hierarchical Goal+Task Planner Implemented in Python

Dana Nau,[1,2] Yash Bansod,[2] Sunandita Patra,[1] Mark Roberts,[3] Ruoxi Li[1]

[1]Dept. of Computer Science and [2]Institute for Systems Research, U. of Maryland, College Park, MD, USA
[3]The U.S. Naval Research Laboratory, Code 5514, Washington, DC, USA
[1,2]{nau, yashb, patras, rli12314}@umd.edu, [3]{first.last}@nrl.navy.mil

Abstract

The Pyhop planner, released in 2013, was a simple SHOP-style planner written in Python. It was designed to be easily usable as an embedded system in conventional applications such as game programs. Although little effort was made to publicize Pyhop, its simplicity, ease of use, and understandability led to its use in a number of projects beyond its original intent, and to publications by others.

GTPyhop (Goal-and-Task Pyhop) is an extended version of Pyhop that can plan for both goals and tasks, using a combination of SHOP-style task decomposition and GDP-style goal decomposition. It provides a totally-ordered version of Goal-Task-Network (GTN) planning without sharing and task insertion. GTPyhop's ability to represent and reason about both goals and tasks provides a high degree of flexibility for representing objectives in whichever form seems more natural to the domain designer.

1 Introduction

Pyhop[1] is a simple HTN planner written in Python, comprising less than 150 lines of Python code. Its planning algorithm is based on the one in SHOP (Nau et al. 1999), but it avoids the need for a specific "planning" language by having the task network and its methods written directly in Python. Pyhop's development was motivated by an observation that application developers were often writing planning systems themselves, rather than learning specialized AI planning languages (Nau 2013). Pyhop was written in hopes of providing an HTN (Hierarchical Task Network) planner that could be easily understood by non-AI practitioners.

Pyhop's author made little effort to publicize it, but the ease with which it could be understood and used has made it useful for rapid prototyping, leading to its use in a number of projects and publications by others (see Section 2).

This paper describes GTPyhop, which extends Pyhop to plan for goals as well as tasks. It combines aspects of both HTN planning as in Pyhop and SHOP, and HGN planning as in GDP (Goal Decomposition Planner) (Shivashankar et al. 2012). In the terminology of (Alford et al. 2016b), it does a totally-ordered version of GTN planning without sharing and task insertion (there is an example at the end of Section 3).

GTPyhop's source code is about four times as big as Pyhop's. It includes the following features:

[1]https://bitbucket.org/dananau/pyhop/src/master/

- Rather than a task list, GTPyhop has a *to-do* list that contains zero or more actions, tasks, and goals. Like Pyhop, it decomposes tasks using task methods; and like GDP, it decomposes goals using goal methods. However, all methods return to-do lists, rather than Pyhop's task lists or GDP's goal lists (see example at end of Section 3). Thus a planning domain may use any arbitrary combination of task decomposition and goal decomposition.
- Since HGN planning semantics corresponds readily to classical goal semantics (Shivashankar et al. 2012), it can be used to guarantee soundness. To enforce soundness, when GTPyhop decomposes a goal g, it verifies whether the resulting plan actually accomplishes g, and backtracks if g is not accomplished. We anticipate that in future work, this may be useful for purposes such as verification and validation of domain descriptions.
- GTPyhop can load multiple planning domains into memory, and switch among them without having to restart Python each time. GTPyhop also includes more documentation than Pyhop, and additional debugging features.

The GTPyhop software distribution is available for download under an open-source license.[2] In addition to GTPyhop, it includes several example domains, a test harness for running them, and a simple example of planning-and-acting integration: a version of the Run-Lazy-Lookahead actor (Ghallab, Nau, and Traverso 2016) that uses GTPyhop as its planner.

The paper is organized as follows. We provide some context for the original version of Pyhop (Section 2), describe GTPyhop (Section 3), and briefly describe several research projects in which GTPyhop is being used and extended (Section 4). This is followed by discussions of related work (Section 5), some of GTPyhop's limitations (Section 6), and concluding remarks (Section 7).

2 Why does Py Hop?

As we mentioned earlier, Pyhop is basically a simplified version of SHOP that uses Python syntax. For example, actions and methods are written directly as Python functions. Their preconditions are Python *if* tests, and their effects are their returned values.

[2]https://github.com/dananau/GTPyhop

According to (Nau 2013), the original motivation for Pyhop was a workshop on AI in games (Lucas et al. 2012) in which many of the attendees were developing games that incorporated AI planners as subsystems. The approach was like the way AI planning has been used in other systems that operate in dynamically changing environments:

- Approximate a part of the system's objective as a planning problem p, and develop a special-purpose planner for p.
- Use the planner as a subroutine, calling it repeatedly to replan as the world changes. The planner operates online, tightly integrated with the rest of the system.

Such integration is easier if the planner is small, easily understandable, and uses data structures compatible with those used in the larger system in which the planner is embedded, rather than requiring the data to be translated between two different representation schemes. Pyhop was written as an example of how to facilitate such integration.

The only efforts to publicize Pyhop were a brief announcement on the SHOP web site[3] and an invited workshop talk (Nau 2013) with no published paper, just slides. Despite this, a Google Scholar search[4] shows 66 publications that refer to Pyhop. In many of them, Pyhop is used for applications having nothing to do with games. There also have been several forks of Pyhop, e.g., (McGreggor 2014; Cheng et al. 2018), and a re-implementation of Pyhop in C++ (Jacopin 2020).

3 GTPyhop

Figure 1 shows the GTPyhop planning algorithm. Like Pyhop, it does a depth-first search with no loop detection (which wouldn't be useful here, see Alford et al. (2012)). Note that:

1. apply-action-and-continue and refine-task-and-continue handle actions and tasks the same way that Pyhop does.
2. The way refine-goal-and-continue decomposes goals is based on GDP (Shivashankar et al. 2012).
3. A mixture of task decomposition and goal decomposition may be used throughout a planning domain. In the to-do lists T and T_{sub} in lines (*i*), (*ii*), and (*iii*), each element may be a task, goal, or action.
4. In line (*iii*), g is a goal, so GTPyhop needs to ensure that T_{sub} achieves g. To do so, it appends to T_{sub} a dummy action *verify*(g) that has g as a precondition. If the state produced by T_{sub} satisfies g, the action has no effect. Otherwise the action fails, making GTPyhop backtrack.

```
GTPyhop($s_0, T$)                                                    (i)
  return seek-plan($s_0, T, [\,]$)  # base case for seek-plan

seek-plan($s, T, \pi$)
  # recursive DFS; $\pi$ is the current partial solution
  if $T = [\,]$ then return $\pi$
  $t \leftarrow$ the first element of $T$
  $T' \leftarrow$ the rest of $T$
  case($t$):    # solve $t$, then plan for $T'$
    action: return apply-action-and-continue($s, t, T', \pi$)
    task: return refine-task-and-continue($s, t, T', \pi$)
    goal: return refine-goal-and-continue($s, t, T', \pi$)

apply-action-and-continue($s, a, T', \pi$)
  if action $a$ is applicable in state $s$:
    return seek-plan($\gamma(s, a), T', \pi + [a]$)
  else: return failure

refine-task-and-continue($s, \tau, T', \pi$)
  $M \leftarrow \{$task-methods that were declared relevant for $\tau\}$
  for each $m \in M$ that is applicable in $s$:
    $T_{\text{sub}} \leftarrow$ decomp($s, m$)                                  (ii)
    $\pi \leftarrow$ seek-plan($s, T_{\text{sub}} + T', \pi$)
    if $\pi \neq$ failure then return $\pi$
  return failure

refine-goal-and-continue($s, g, T', \pi$)
  $M \leftarrow \{$goal-methods that were declared relevant for $g\}$
  for each $m \in M$ that is applicable in $s$:
    $T_{\text{sub}} \leftarrow$ decomp($s, m$) + [verify($g$)]                 (iii)
    $\pi \leftarrow$ seek-plan($s, T_{\text{sub}} + T', \pi$)
    if $\pi \neq$ failure then return $\pi$
  return failure
```

Figure 1: GTPyhop pseudocode. The arguments are the initial state s_0 and a to-do list T (a list of actions, tasks, and goals). GTPyhop returns a solution plan π, or failure if no solution exists. $\gamma(s, a)$ is the state produced by a, and decomp(s, m) is the to-do list produced by m. "+" is concatenation of lists.

3.1 Representations and examples

We now describe the basic elements of a GTPyhop planning domain, with examples from the GTN blocks-world domain included with the GTPyhop software distribution.[5]

Domains are Python objects that contain all the elements of a planning domain, e.g., gtpyhop.Domain('blocks_gtn').

States are Python objects that contain collections of state-variable bindings. When one first defines a state s, it is easiest to write the variable bindings in dictionary form, as in Figure 2. Afterward, one can use a Python version of state-variable notation, e.g., s.pos[x] = 'hand' in Figure 3.

Actions and methods are Python functions, with the current state as the first argument. There isn't a special reasoning system (e.g., SHOP's Horn-clause inference) to evaluate an action's or method's preconditions. Instead, preconditions are Python *if* tests. Similarly, an action's effects and a method's to-do list are produced by ordinary Python computations.

For example, Figure 3 shows the blocks-world pickup action. Its arguments are the current state s and the block x to pick up. If the precondition (the first *if* test) is satisfied, the action modifies the state to say that x is in the robot hand, and returns the modified state. Otherwise the action returns no value, which tells GTPyhop the action is inapplicable.

Tasks and task methods: Tasks are written as Python tuples. For example, let ('take',x) be the task of picking up a block

[3]http://www.cs.umd.edu/projects/shop/

[4]https://scholar.google.com/scholar?hl=en&q=pyhop

[5]https://github.com/dananau/GTPyhop/tree/main/Examples/blocks_gtn

```
sus_s0 = gtpyhop.State('Sussman initial state')
# Python dictionary notation for sus_s0.pos['a'] = 'table', etc.
sus_s0.pos = {'a':'table', 'b':'table', 'c':'a'}
sus_s0.clear = {'a':False, 'b':True, 'c':True}
sus_s0.holding = {'hand':False}

sus_sg = gtpyhop.Multigoal('Sussman goal')
sus_sg.pos = {'a':'b', 'b':'c'}
```

Figure 2: GTPyhop version of the Sussman anomaly (Ghallab, Nau, and Traverso 2004, Section 4.4). In the initial state sus_s0, blocks a and b are on the table, and block c is on a. The goal sus_sg specifies that a is on b, and b is on c.

```
def pickup(s,x):
    if s.pos[x] == 'table' and s.clear[x] == True \
        and s.holding['hand'] == False:
        s.pos[x] = 'hand'
        s.clear[x] = False
        s.holding['hand'] = x
        return s
gtpyhop.declare_actions(pickup)
```

Figure 3: The blocks-world pickup action. The arguments are the current state s and a block x. If the precondition (the *if* test) is satisfied, the action modifies s and returns it. The last line declares pickup to be an action.

```
def m_take(s,x):
    if s.clear[x] == True:        # precondition
        if s.pos[x] == 'table':    # decide what to do
            return [('pickup', x)]
        else: return [('unstack', x, s.pos[x])]
gtpyhop.declare_task_methods('take',m_take)
```

Figure 4: A task method. Its arguments are the current state s and a block x. If the precondition is satisfied, it returns a to-do list containing a pickup action if x is on the table, or an unstack action if x is on a block. The last line declares m_take to be relevant for all tasks of the form (take, ...).

x that may be either on the table or a block. Figure 4 shows a method for this task. If there are several methods for the same task, GTPyhop (like Pyhop and SHOP) will try them in the order that they occur in the source file.

Goals can be represented in two ways. A *unigoal* is a triple that represents a desired value for a state variable, e.g., ('pos', 'a', 'b') is the goal of reaching any state s such that s.pos['a']=='b'. A *multigoal* is a state-like object that represents a conjunction of unigoals, e.g., sus_sg in Figure 2 represents the conjunction of ('pos', 'a', 'b') and ('pos', 'b', 'c').

Goal methods include *unigoal methods* for decomposing unigoals, and *multigoal methods* for decomposing multigoals. Figure 5 shows a multigoal method that implements a near-optimal block-stacking algorithm. For example, find_plan(sus_s0,sus_sg) returns the following plan:

```
def m_moveblocks(s, mgoal):
    for x in all_clear_blocks(s):   # find a block to move
        stat = status(x, s, mgoal)
        if stat == 'move-to-block':
            where = mgoal.pos[x]   # where to move it
            return [('take',x), ('put',x,where), mgoal]      (iv)
        elif stat == 'move-to-table':
            return [('take',x), (put,x,'table'), mgoal]      (v)
    for x in all_clear_blocks(s):   # resolve deadlock
        if status(x, s, mgoal) == 'waiting'  \
            and s.pos[x] != 'table':
            return [('take',x), ('put',x,'table'), mgoal]    (vi)
    return [ ]     # no blocks need to be moved
gtpyhop.declare_multigoal_methods(m_moveblocks)
```

Figure 5: A multigoal method that implements the block-stacking algorithm in (Gupta and Nau 1992): if a block needs moving and is ready to move to its final location, then do so and continue planning for mgoal; else if there's a deadlock then resolve it and continue planning for mgoal; else we're done. There are two helper functions: all_clear_blocks returns a list of clear blocks, and status tells whether a block x needs to be moved and whether it is ready to be moved.

```
[('unstack', 'c', 'a'), ('putdown', 'c'), ('pickup', 'b'),
 ('stack', 'b', 'c'), ('pickup', 'a'), ('stack', 'a', 'b')]
```

A GTN planning example. In lines (*iv*), (*v*), and (*vi*) of Figure 5, the to-do lists contain two tasks and a multigoal. GTPyhop (Figure 1) uses the right kind of method for each.

4 Example Usages

There are several research projects in which GTPyhop is being used and extended. We briefly describe them below.

Bansod et al. (2021) describes an integrated system for hierarchical planning and acting in dynamically changing environments. An important component of this system is a re-entrant planning algorithm based on GTPyhop.

A paper in preparation integrates GTPyhop with reinforcement learning. The work uses the goal network provided by GTPyhop to guide a curricula for multi-task learning. During acting, it executes the goal network provided by GTPyhop.

GTPyhop is also being used in a research project to develop temporal planning algorithms in multi-agent environments. For this purpose, modifications are being made to support communication among multiple agents, and representation and reasoning about temporal constraints.

In our future work, we anticipate the possibility of using goals for verification and validation of domain descriptions.

5 Related Work

The closest related theoretical work is (Alford et al. 2016b), which related task networks and goal networks under various semantics, including HTN, HGN, and GTN planning, task (or goal) insertion, and sharing.

Several HTN planners introduced in the 1970s through 1990s are no longer available for comparison. One of the first was NOAH (Sacerdoti 1975), which was followed by Nonlin (Tate 1977), the SIPE family (Wilkins 1990), O-Plan (Currie and Tate 1991; Tate, Drabble, and Kirby 1994), PRS (Ingrand et al. 1996; Meyers 2016), and UMCP (Erol 1996).

Many HTN planners provide a planner-specific language in which to write the HTN methods. The SHOP planners (Nau et al. 1999, 2003; Goldman and Kuter 2019) make use of Lisp's extensibility to define a Lisp-like language for this purpose. Sohrabi et al. (2009) extended PDDL3 with HTNs to support preferences and converted this extended PDDL3 format for a variant of SHOP2.

In JSHOP2 (Ilghami and Nau 2003), methods are written in the same Lisp-like language, but JSHOP2 compiles them to Java to perform the search. Similarly, the HyperTensioN planner converts a planning model into the Ruby language and was recently extended to support semantic attachments for HTN (Magnaguagno and Meneguzzi 2020).

The totSAT planner (Behnke, Höller, and Biundo 2018) converts totally-ordered HTN planning problems into a SAT formula. PANDA (Höller et al. 2021) is a planner that integrates various approaches to hierarchical planning. Both planners emphasize the importance of a common language for problem definition and propose a Hierarchical Domain Definition Language (HDDL) for it (Höller et al. 2020).

HDDL was also the input language for the recent Hierarchical Track of the International Planning Competition.[6] A full discussion of all planners in that competition is out of scope for a short paper, but we highlight the winners SIADEX (de la Asunción et al. 2005; Castillo et al. 2005) and HyperTensioN (Magnaguagno and Meneguzzi 2020), which both translated HDDL to their specific format.

IxTeT (Ghallab and Laruelle 1994) was a temporal HTN planner which used a specialized language to encode its methods. OpenPRS[7] is a C implementation of PRS. ASPEN (Fukunaga et al. 1997; Chien et al. 2000) is both a planner and framework for planning in space applications; it uses a planner-specific language for encoding plans. FAPE (Dvorák et al. 2014; Bit-Monnot et al. 2020) is a recent planner that supports a subset of the ANML language (Smith, Frank, and Cushing 2008). A more recent HTN planner, SHPE (Menif, Jacopin, and Cazenave 2014), has been specifically developed for AI planning in video games. It uses a simplified variant of ANML (Smith, Frank, and Cushing 2008) to encode problems that are compiled into C++ to perform search. The Adversarial HTN Planner (Ontañón and Buro 2015) allows for HTNs to be used in iterated environments such as Real Time Strategy games; problems are encoded in a language provided by the system. A variety of research has extended this planner (e.g., (Lin et al. 2020; Sun et al. 2017)).

Like Pyhop and GTPyhop, there are several HTN planners in which domains and problems are written in a conventional programming language. The planner by Neufeld et al. (2018) uses C++ for this purpose; its HTN primitives link with Behavior Trees, a common representation for computer game agents. The planner by Soemers and Winands (2016) also uses C++ to represent HTN problems; this planner introduced a mechanism to reuse the existing solution for faster replanning. The UPOM planner introduced in (Patra et al. 2020) uses Python to represent hierarchical operational models.

[6]http://gki.informatik.uni-freiburg.de/competition/

[7]https://git.openrobots.org/projects/openprs

6 Limitations

One limitation involves the goal representation's expressivity. A GTPyhop goal, like a state (see Figure 2), is a set of state-variable bindings. The goal is the conjunction of those bindings, without a way to represent more complicated logical expressions. There probably are some workarounds, but we have not yet considered this. In our work so far, this limitation has not been a major problem.

In many HTN planners, a method or action may contain free variables for which there are several possible instantiations. When the planner creates instances of the method or action, it may backtrack over these instantiations. In contrast, in GTPyhop (like Pyhop) there is no notion of instantiating a method. The method is a piece of Python code that GTPyhop calls directly. The method may contain a variety of local variables, but it is up to the method's author to specify how these variables will acquire their values.

In HTN planners that use planner-specific languages, the methods' preconditions and subtasks are data structures that the planner can reason about before deciding which methods to use in a planning problem. This has enabled several recent advances in HTN-planning search heuristics (Alford et al. 2016a) and other speedup techniques (Behnke, Höller, and Biundo 2018). In contrast, GTPyhop does not know its methods' preconditions and subtasks in advance, because each method is a Python program that *computes* a list of subtasks and subgoals that may depend on the current state (e.g., Figures 4 and 5). A potential way to circumvent this limitation might be to evaluate the method, see what tasks, goals, and actions it returns, and *then* use this information to provide input to a search heuristic—but we have not tried to implement this to see how well it would work. For now, GTPyhop (like Pyhop and SHOP) just does a depth-first search, trying methods in the order that the domain author defined them.

7 Conclusions

Pyhop implemented a version of SHOP-style HTN planning in which methods and actions were written directly in Python. Despite a minimal amount of publicity and no publication, it has been used in several systems that went beyond its original intent of a simple planner for game systems.

GTPyhop extends Pyhop to provide a version of totally-ordered Goal-Task-Network planning without sharing and task insertion. GTPyhop also includes several other features, as described in the introduction. We are working now to extend GTPyhop to incorporate temporal and multi-agent concerns. Section 4 has briefly described the directions that this work is taking.

Acknowledgments. Many thanks to the anonymous reviewers for their insightful remarks. This work has been supported in part by ONR grant N000142012257 and NRL grants N0017320P0399 and N00173191G001. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

References

Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. W. 2016a. Bound to plan: exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *ICAPS*.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2012. HTN problem spaces: Structure, algorithms, termination. In *SOCS*.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016b. Hierarchical planning: relating task and goal decomposition with task sharing. In *IJCAI*, 3022–3028.

Bansod, Y.; Nau, D. S.; Patra, S.; and Roberts, M. 2021. Refinement Acting vs. Simple Execution Guided by Hierarchical Planning. In *ICAPS Workshop on Hierarchical Planning (HPlan)*.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – totally-ordered hierarchical planning through SAT. In *AAAI*.

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. E. 2020. FAPE: a constraint-based planner for generative and hierarchical temporal planning. In *arXiv:2010.13121*.

Castillo, L.; Fdez-Olivares, J.; García-Pérez, Ó.; and Palao, F. 2005. Temporal enhancements of an HTN planner. In *Conf. Spanish Assoc. for Artificial Intelligence*, 429–438.

Cheng, K.; Wu, L.; Yu, X.; Yin, C.; and Kang, R. 2018. Improving HTN planning performance by the use of domain-independent heuristic search. *Knowledge-Based Systems* 142: 117–126.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; Smith, B.; Fisher, F.; Barrett, T.; Stebbins, G.; and Tran, D. 2000. ASPEN - automating space mission operations using automated planning and scheduling. In *SpaceOps 2000*.

Currie, K.; and Tate, A. 1991. O-Plan: the open planning architecture. *Artificial Intelligence* 52(1): 49–86.

de la Asunción, M.; Castillo, L.; Fdez-Olivares, J.; García-Pérez, O.; González, A.; and Palao, F. 2005. SIADEX: An interactive knowledge-based planner for decision support in forest fire fighting. *AI Communications* 18(4): 257–268.

Dvorák, F.; Barták, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. planning and acting with temporal and hierarchical decomposition models. In *ICTAI*, 115–121.

Erol, K. 1996. *Hierarchical task network planning: formalization, analysis, and implementation*. Ph.D. thesis, University of Maryland.

Fukunaga, A. S.; Rabideau, G.; Chien, S.; and Yan, D. 1997. ASPEN: A Framework for Automated Planning and Scheduling of Spacecraft Control and Operations. In *ISAIRAS*.

Ghallab, M.; and Laruelle, H. 1994. Representation and Control in IxTeT, a Temporal Planner. In *AIPS*, 61–67.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical task network planning in Common Lisp: the case of SHOP3. In *Proc. European Lisp Symposium*, 73–80.

Gupta, N.; and Nau, D. S. 1992. On the Complexity of Blocks-World Planning. *Artificial Intelligence* 56(2-3): 223–254.

Höller; Behnke; Bercher; and Biundo. 2021. The PANDA framework for hierarchical planning. *KI-Künstliche Intelligenz* 1–6.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: an extension to PDDL for expressing hierarchical planning problems. In *AAAI*.

Ilghami, O.; and Nau, D. S. 2003. A general approach to synthesize problem-specific planners. Technical Report CS-TR-4597, UMIACS-TR-2004-40, University of Maryland.

Ingrand, F. F.; Chatila, R.; Alami, R.; and Robert, F. 1996. PRS: A high level supervision and control language for autonomous mobile robots. In *ICRA*, 43–49. IEEE.

Jacopin, E. 2020. Simple-HTN-Planner. GitHub. URL https://github.com/PCfVW/Simple-HTN-Planner.

Lin, S.; Anshi, Z.; Bo, L.; and Xiaoshi, F. 2020. HTN Guided Adversarial Planning for RTS Games. In *ICMA*, 1326–1331.

Lucas, S. M.; Mateas, M.; Preuss, M.; Spronck, P.; and Togelius, J., eds. 2012. *Artificial and Computational Intelligence in Games*, volume 6. Schloss Dagstuhl.

Magnaguagno, M. C.; and Meneguzzi, F. 2020. Semantic Attachments for HTN Planning. In *AAAI*, 9933–9940.

McGreggor, D. 2014. PyHOP, version 2.0. GitHub. URL https://github.com/oubiwann/pyhop.

Menif, A.; Jacopin, É.; and Cazenave, T. 2014. SHPE: HTN planning for video games. In *Workshop on Computer Games*, 119–132.

Meyers, K. L. 2016. A Procedural Knowledge Approach to Task-level Control. In *AIPS*, 158–165.

Nau, D. S. 2013. Game applications of HTN planning with state variables. In *ICAPS 2013 Workshop on Planning in Games*. URL https://cs.umd.edu/~nau/papers/nau2013game.pdf. Invited talk.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20: 379–404.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: simple hierarchical ordered planner. In *IJCAI*, 968–973.

Neufeld, X.; Mostaghim, S.; and Brand, S. 2018. A hybrid approach to planning and execution in dynamic environments through HTNs and behavior trees. In *AIIDE*.

Ontañón, S.; and Buro, M. 2015. Adversarial HTN Planning for Complex Real-Time Games. In *IJCAI*, 1652–1658.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. S. 2020. Integrating acting, planning, and learning in hierarchical operational models. In *ICAPS*, 478–487.

Sacerdoti, E. 1975. The Nonlinear Nature of Plans. In *IJCAI*.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *AAMAS*, 981–988.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML Language. In *ICAPS KEPS Workshop*.

Soemers, D. J. N. J.; and Winands, M. H. M. 2016. HTN Plan Reuse for video games. In *CIG*, 1–8.

Sohrabi, S.; Baier, J. A.; and McIlraith, S. A. 2009. HTN Planning with Preferences. In *IJCAI*, 1790–1797.

Sun, L.; Jiao, P.; Xu, K.; Yin, Q.; and Zha, Y. 2017. Modified Adversarial HTN Planning in RTS Games. *Applied Sciences* 7(9).

Tate, A. 1977. Generating project networks. In *IJCAI*, 888–893.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-Plan2: an open architecture for command, planning and control. In Zweben, M.; and Fox, M. S., eds., *Intelligent Scheduling*. Morgan Kaufmann.

Wilkins, D. E. 1990. Can AI planners solve practical problems? *Computational intelligence* 6(4): 232–246.