A Formal Analysis of Hierarchical Planning with Multi-Level Abstraction

Michael Staud[1,2]

[1]StaudSoft UG, D-88213 Ravensburg, Germany
[2]Ulm University, Institute of Artificial Intelligence, James-Franck-Ring, 89081 Ulm

Abstract

We analyze Hierarchical World State Planning (HWSP), a novel algorithm that tackles the scalability limitations of Hierarchical Task Network (HTN) planning. By combining multi-level state abstraction with predictive task decomposition, HWSP reduces exponential search space growth. We formalize predictive separable tasks, classify planning domains, and derive tight bounds on search complexity. Our results show that HWSP transforms exponential interactions into linear coordination, enabling near-linear top-level planning effort in favorable domains. This enables efficient planning in complex domains that were previously intractable, opening up new possibilities for real-world applications.

Keywords

Hierarchical World State Planning, HTN Planning, Multi-Level Abstraction, Predictive Task Decomposition, Domain Classification, Search Complexity Analysis, Scalability in AI Planning, Backtracking Behavior, Separable Abstract Tasks, Probabilistic Prediction Models

1. Introduction

Complex real-world domains, such as urban planning or logistics problems, are challenging due to their scale and intricacy. One solution is to use a Hierarchical Task Network (HTN) planning approach to reduce the size of the search space. However, these problems are often still too large—both in memory and computational demands—to be handled by today's planners. One new approach is the Hierarchical World State Planning (HWSP) algorithm [1], which reduces the search space by abstracting the world state. The problem is then divided into smaller, manageable subproblems.

In this paper, we provide a theoretical analysis of HWSP's planning efficiency across varying domain structures. We introduce a classification of domains – EX1, EX2, and EX3 – that captures how separable tasks interact and when backtracking is necessary. This classification yields new insights into the structural properties that affect planning complexity.

Our results establish tight upper bounds on the number of states visited during search under each domain class. We contrast these results with classical HTN planning, showing that under certain assumptions (e.g., sibling task independence), HWSP transforms the exponential interaction between sibling tasks into linear coordination, while preserving the necessary exponential search within each separable task's local scope. Furthermore, we incorporate a probabilistic analysis of binary decomposition predictors, showing how predictor quality impacts search effort in EX2 domains.

By grounding HWSP in a rigorous formal framework and connecting it to classical HTN planning theory, we provide a new foundation for understanding the complexity of hierarchical planning. Our findings demonstrate that HWSP can offer substantial improvements in planning efficiency by exploiting structural domain properties and leveraging prediction mechanisms.

We organize the paper as follows: After reviewing related work in Section 2, Section 3 introduces the formal framework of Hierarchical World State Planning (HWSP), including its planning model, semantics, and interface definitions. We also present a city planning domain that is used to illustrate the key concepts. In Section 4, we define three domain classes (EX1 – EX3) that constrain how separable

CAIPI25: ECAI Workshop on AI-based Planning for Complex Real-World Applications, October 25-26, 2025, Bologna, Italy
EMAIL: michael.staud@uni-ulm.de (M. Staud)
URL: https://www.staudsoft.com (M. Staud)

abstract tasks behave, particularly with respect to decomposition and backtracking. Section 5 analyzes the backtracking behavior of HWSP under these domain classes. Section 6 presents the theoretical results comparing HWSP and classical HTN planning.

2. Related Work

Early work by Korf [2] demonstrated that decomposing a planning problem into subproblems can yield significant improvements in search efficiency. This idea was further developed by Knoblock [3], who introduced algorithms for automatically creating abstraction hierarchies and proved that solving problems on multiple levels of abstraction can be exponentially more efficient than flat planning. The concept of factored planning, introduced by Brafman and Domshlak [4], also shows that problem factorization can lead to exponential reductions in effort.

Macro-operators have also been recognized as a powerful tool for improving HTN planning efficiency. Korf's seminal paper [5] on macro-operators showed that learning effective macro-operators can reduce the effective solution depth and transform planning processes with exponential complexity into polynomial ones. More recent work by Botea et al. [6] demonstrated the effectiveness of automatically learned macro-operators in improving planning performance.

Hierarchical reinforcement learning (HRL) has also been explored as a means of improving planning performance. Dietterich's foundational paper from 2000 introduced the MAXQ framework [7], which formalizes how a Markov Decision Process (MDP) can be decomposed into a temporal hierarchy of subtasks. More recent work by Kulkarni et al. [8] established that HRL can achieve state-of-the-art performance on complex game environments.

Backtracking and planning dynamics in HTNs have also been extensively studied. Nau et al.'s work [9] examined the impact of task ordering on HTN search, while Olz and Bercher [10] introduced a look-ahead technique for reducing backtracking in HTN plan search. Commitment strategies in HTN planning have also been explored by Tsuneto et al. [11], who compared different strategies for variable and method commitment. Theoretical analyses by Alford et al. [12] and Geier and Bercher [13] have provided valuable insights into the complexity of HTN planning and the effects of task ordering on search efficiency. McCluskey [14] introduced EMS, which uses object transition sequences and sort-based independence to enable local reasoning about how dynamic objects change state during hierarchical planning.

Beyond classical HTN planning, several foundational and conceptually related approaches have addressed hierarchical abstraction. Sacerdoti's ABSTRIPS system [15] and Knoblock's work on automatic abstraction hierarchy construction [16] pioneered world-state abstraction in planning, demonstrating that solving abstract problems first can drastically reduce search. Angelic hierarchical planning [17] further introduced optimistic high-level actions with abstract effect summaries, similar to separable abstract tasks but emphasizing cost bounds for optimization rather than multi-level abstraction for scalability. Recent work has also explored compiling HTNs to SAT for optimal search [18], defining standard hierarchical languages like HDDL [19], and integrating HTN planning into competitive evaluations via the IPC track.

3. Framework Definitions

This Section describes the Hierarchical World State Planning (HWSP) algorithm, which extends the traditional planning paradigm by introducing a multi-level world state [20, 1]. The HWSP algorithm is based on the concept of *separable abstract tasks*, which enable more efficient planning by partitioning the planning process into smaller sub-processes across multiple levels of abstraction. The following version of the HWSP algorithm is a more formal one compared to the original paper [21, 22]. In this framework, we assume a progressive planning strategy in which decomposition proceeds front-to-back: tasks are handled in their plan order, and the world state is updated strictly forward from the initial state.

Let C, V, P and A denote the set of all constants, variables, predicates and atoms, respectively. A literal is an atom or its negation, while an atom is a predicate applied to a tuple of terms. Here, a term may be a constant or a variable. Every predicate p belongs to the set P.

3.1. Example: City Planning

We introduce a running example to help illustrate the concepts of this paper. This domain models a hierarchical city planning problem with three levels of detail (abstraction): *City*, *Quarter*, and *Building*. Streets are not explicitly modeled. Each level consists of a 2D grid of square cells. A quarter consists of a 4×4 block of building cells, and a city consists of a 4×4 block of quarters – forming a non-overlapping, nested hierarchy (see Figure 1).

Some cells may be *blocked*, preventing construction; this is represented by the predicate $\mathsf{blocked}(x)$, which holds in the initial state for blocked cells. Each level contains objects that can be constructed via tasks:

- **City Level:** *small-city*, *big-city*
- **Quarter Level:** *industrial-quarter*, *residential-quarter*, *park*
- **Building Level:** *living-house*, *playground*, *football-field*, *powerplant*, *factory*

To demonstrate the algorithm's ability to handle different domain classes, we can modify this base domain in three key ways. First, we can introduce blocked cells, which may cause tasks to fail and lead to different planning outcomes. Second, we can enable electricity constraints, where a single `powerplant` – placed in any quarter – can power the entire city. Third, power propagation is expressed explicitly through the world state: a quarter is considered powered if its city contains a powerplant, captured by a specific atom in the world state.

3.2. Planning Domain

A planning domain is denoted as $D = (T_a, T_p, M, L, L_d, D_{\text{der}}, \Phi)$ where T_a is the set of all abstract tasks (including both standard and separable tasks), $T_S \subset T_a$ denotes the set of separable abstract tasks, T_p is the set of primitive tasks, M is the set of methods, and $L \in \mathbb{N}$ the number of detail levels (i.e., abstraction levels) present in the domain. The detail level function $L_d : P \cup T_a \cup T_p \to \mathbb{N}$ assigns each predicate or task a detail level ℓ. The derived predicates are defined in D_{der}, with their corresponding logical formulas specified in Φ.

Both abstract and primitive tasks are tuples in the form $t(\tau) = \langle prec_t(\tau), \mathit{eff}_t(\tau) \rangle$, where each task $t(\tau) \in T_a \cup T_p$ has a precondition $prec_t(\tau)$ and an effect $\mathit{eff}_t(\tau)$. For separable abstract tasks $t \in T_S$, we write $\mathit{eff}_t(\tau) = \mathit{meff}_t(\tau) \cup \mathit{peff}_t(\tau)$ where meff_t are mandatory effects and peff_t are predicted (non-guaranteed) effects. In the planning process, these are predicted by the prediction function. A plan step is a uniquely labeled task $l : t(\overline{\tau})$.

Example The task *residential-quarter* has mandatory effects meff_t such as "provides housing capacity" (which must be achieved for the quarter to be valid). It could have predicted effects peff_t such as "increases local property values".

A *method* is a tuple $m = \langle t_a(\overline{\tau}_m), P_m \rangle$, where $t_a(\overline{\tau}_m)$ is the abstract task it can decompose, and P_m is a series of plan steps (total-ordered), with $\overline{\tau}_m$ as the parameters of the method.

A *state* s is an element of $\mathcal{P}(A)$, where $\mathcal{P}(A)$ denotes the power set of A. We require that a separable abstract task t_S at detail level ℓ decomposes only into tasks at detail level $\ell + 1$, while other tasks decompose at the same level ℓ. Hence, separable tasks must occur at levels $\ell < L$. Let $s \downarrow \ell = \{a \in s | L_d(a) = \ell\}$ denote the projection of the state s to the detail level ℓ.

Example In our domain, we have $L = 3$ detail levels which includes city (level 1), quarter (level 2), and building (level 3). Separable abstract tasks T_S include *small-city*, *big-city* at level 1, and *residential-quarter*, *industrial-quarter*, *park* at level 2. The restrictions on the methods regarding levels means that a city task can only create quarters not buildings directly.

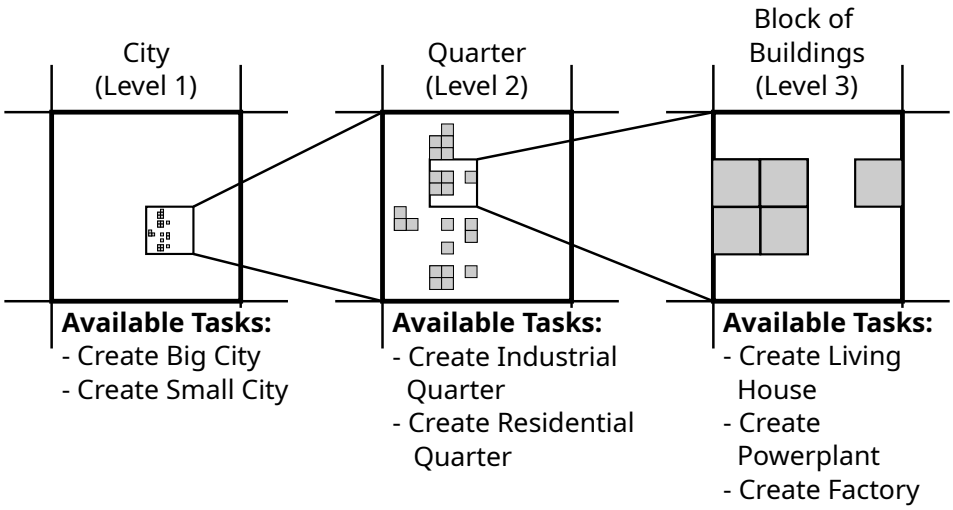

Figure 1: The example domain used to illustrate the HWSP algorithm.

Let $D_{\text{der}} \subset P$ be the set of derived predicates [23], which includes all predicates that are not on the most concrete (non-abstract) detail level. For each derived predicate $P_{\text{der}} \in D_{\text{der}}$ at detail level ℓ, there exists a logical formula $\phi_{P_{\text{der}}} \in \Phi$ involving only predicates from the next detail level ($\ell + 1$). Formally:

$$P_{\text{der}}(x_1, ..., x_n) \equiv \phi_{P_{\text{der}}}(p_{\ell+1}(x_1), ..., p_{\ell+1}(x_n))$$

where:

- $p_{\ell+1}$ represents a predicate at detail level $\ell + 1$,
- $x_1, ..., x_n$ are variables representing objects or entities in the domain,
- $\phi_{P_{\text{der}}}$ is a first-order logic formula that combines the predicates from detail level $\ell+1$ using logical operators (e.g., $\wedge$, $\vee$, $\neg$).

Each derived predicate P_{der} at detail level ℓ can only depend on predicates that are strictly at level $\ell + 1$ (this is more restricted than in PDDL 2.2). Furthermore, no cycles are allowed, even across multiple predicates.

Example Derived predicates in our examples are the predicates on the city and quarter level. On the quarter level *has_power(quarter)* is in a world state if the corresponding sub-rectangle at the next detail level (building level) contains a building classified as a power plant.

We restrict our attention to domains with a finite decomposition depth. The decomposition depth represents the longest chain of tasks achievable through method application. The maximum decomposition depth in any domain D, denoted $D_{\max}(D)$, is assumed to be less than or equal to Δ. Every domain considered in this paper satisfies $D_{\max}(D) \leq \Delta$ for a fixed but arbitrary natural number Δ. A **planning problem** is defined as a pair $\Pi = (s_0, t_0)$, where $s_0 \in \mathcal{P}(A)$ is the initial world state, and $t_0 \in T_a$ is the initial abstract task to be decomposed.

To enable efficient decomposition, HWSP incorporates predictor functions [1] that estimate separable abstract task outcomes.

Definition 1 (Predictor functions)**.** *HWSP can use learned estimators that operate on separable abstract tasks. Given the current detail level ℓ state $s \downarrow \ell$ and a separable task $t_S \in T_S$ the predictor function returns*

$$\pi(s, t_S) = \big(\varepsilon(s, t_S), \sigma(s, t_S)\big) \in \mathcal{P}(A) \times \{\textit{true}, \textit{false}\},$$

where

- $\varepsilon : \mathcal{P}(A) \times T_S \to \mathcal{P}(A)$ *is an* effect predictor *that proposes a set of* non-mandatory *effects (an estimate for* $peff_{t_S}$*), and*
- $\sigma : \mathcal{P}(A) \times T_S \to \{true, false\}$ *is the* decomposition predictor *whose output is to decide in a planning process if a separable abstract task can be decomposed or not.*

The predictor can be fixed-effect rules, Conv2D, ResNet or any other approximation function. Predictor functions in HWSP were first introduced and formally described by Staud [1].

3.2.1. HTN plans with labelled occurrences

Definition 2 (HTN plan with decomposition tree)**.** *A* plan (or partial plan) *is a tuple*

$$Q = \langle V, E, r, \lambda_T, \lambda_M, \prec \rangle$$

where

- (V, E, r) *is a finite rooted tree whose root is* $r \in V$*,*
- *every* task node $v \in V_T \subseteq V$ *is labeled by* $\lambda_T(v) \in T_a \cup T_p$
- *every* method node $v \in V_M \subseteq V$ *(the remaining vertices) is labeled by a method* $\lambda_M(v) \in M$*, and*
- *a strict total order* $\prec$ *over the task nodes, which is consistent with the child order of each method node.*

Edges alternate strictly between task and method nodes: if $(u, v) \in E$ *and* u *is a task node, then* v *is a method node and* $\lambda_M(v)$ *must be a method for* $\lambda_T(u)$*; conversely, all children of a method node are task nodes that constitute the method plan steps.*

Definition 3 (Separable task predicate)**.** $\mathsf{is\_sep}(Q, v) := \big(\lambda_T(v) \in T_S\big)$.

Let $\mathsf{state\_at}(Q, v)$ be the world state obtained by applying all effects of task nodes $w \prec v$ that are fully decomposed to the initial state s_0. By Definition 7, all task nodes w such that $w \prec v$ must be fully decomposed before v is considered for decomposition. Hence, the cumulative world state $\mathsf{state\_at}(Q, v)$ is well-defined and includes only effects of primitive tasks. We define:

Definition 4 (Decomposability Test)**.** *Let* Q *be a plan and* $v \in V_T$ *a task node with label* $\lambda_T(v) = t(\bar{\tau})$*. Let* $s := \mathsf{state\_at}(Q, v)$*. Then:*

$$\mathsf{can\_decompose}(Q, v) := true$$

iff there exists a sequence of method applications starting from $t(\bar{\tau})$ *such that: the precondition of every task in the resulting decomposition tree holds in the state at which it is applied, the resulting decomposition tree contains only primitive tasks at the leaves, and all relevant depth and abstraction-level constraints are satisfied.*

Sibling-rewriting relation For two plans Q, Q' and a separable abstract task occurrence $s \in V_T$ of Q we write $Q \rightsquigarrow_s Q'$ iff Q' is obtained from Q by *only* replacing the subtree rooted at s; all other subtrees (including the one currently under investigation) are left unchanged.

Definition 5 (Prior-task predicate)**.** *For any two task occurrences* $v, w \in V_T$ *in a plan* Q*, define*

$$\mathsf{prior}(Q, v, w) := \big(v \prec \cdots \prec w\big),$$

where $\prec$ *is the total order over task nodes that are part of the plan and* $\prec \cdots \prec$ *means that there exists a chain of order relations. In other words,* $\mathsf{prior}(Q, v, w)$ *is* ***true*** *iff the execution of* Q *executes* v *(or some descendant of* v*) before* w*.*

We now define method decomposition under the constraint that planning is progressive. So each plan is guaranteed to have been decomposed in such a way that the current world state can be computed from the initial world state:

Definition 6 (Fully Decomposed Task)**.** *Let $Q = \langle V, E, r, \lambda_T, \lambda_M, \prec \rangle$ be a plan, and let $v \in V_T$ be a task node. We say that v is* fully decomposed *in Q if all leaf task nodes in its descendant subtree rooted at v are primitive tasks, i.e.,*

$$\textit{is\_decomposed}(Q, v) := \forall u \in \textit{descendants}_Q(v) \cap V_T,\ \lambda_T(u) \in T_p.$$

Definition 7 (Progressive Method Decomposition)**.** *Let $Q = \langle V, E, r, \lambda_T, \lambda_M, \prec \rangle$ be a plan and $u \in V_M$ a method node with parent task $t(\bar{\tau}) = \lambda_T(\textit{parent}(u))$. The function* $\mathsf{decompose}(Q, u)$ *returns a new plan Q'* ***only if****:*

1. *All task nodes $w \in V_T$ such that $w \prec t(\bar{\tau})$ (i.e., tasks prior to the parent of u) are* ***fully decomposed****, i.e.:*
$$\forall w \in V_T, (\mathsf{prior}(Q, w, t(\bar{\tau})) \rightarrow \textit{is\_decomposed}(Q, w)).$$
2. *The siblings of $\textit{parent}(u)$ that appear before $\textit{parent}(u)$ in $\prec$ are also fully decomposed.*

This condition ensures that the planner progresses only when all previous tasks are fully resolved into executable steps. The decomposition proceeds as follows:

1. *Add fresh task nodes $s_1, \ldots, s_m$ as children of u (ordered left-to-right).*
2. *Update $\prec$ to replace the parent task's position with $s_1 \prec \cdots \prec s_m$ in a way that preserves the ordering relative to $\textit{parent}(u)$'s siblings.*
3. *If u already has children (idempotence), the plan is unchanged.*

Otherwise, the decomposition is not allowed.

The above definitions will be used in subsequent proofs.

3.3. Planning Process

The most important difference between HTN planning and HWSP is the planning process [1]. In HTN planning, we decompose abstract tasks until only primitive tasks remain. In HWSP, decomposition stops at separable abstract tasks, unless we are on the most detailed level. Instead of decomposing these further, the planner uses a predictor function to estimate their effects and proceeds as if they were primitive tasks at that level. This allows the planner to generate a top-level plan composed of separable abstract tasks, which are expected to work under typical conditions.

For each separable abstract task, a new planning process is launched to generate a subplan that achieves its mandatory effects $\mathit{meff}_t \subseteq D_{\text{der}}$. As the effects are derived predicates, the actual goal is the union of their logical formulas ϕ_{meff_t}. Backtracking is only necessary if this subplan fails. To maintain a consistent world state, planning processes emit one separable abstract task at a time. Informally, each task emitted is treated as a final decision and is only generated after confirming that a (local) plan exists to achieve the required effects. While separable abstract tasks are semantically equivalent to regular abstract tasks, they are treated differently by the planner. Each planning process receives the local world state, $s \downarrow (\ell + 1)$ (projected to the next detail level) as input, and its goal is to achieve the mandatory effects meff_t of the separable abstract task that spawned it.

Definition 8 (Planning Process Solution)**.** *A solution for planning process at level ℓ is a plan Q where every leaf node is either: (1) a primitive task ($\lambda_T(v) \in T_p$) if $\ell + 1 = L$, or (2) a separable abstract task ($\lambda_T(v) \in T_S$) if $\ell + 1 < L$. All preconditions of separable abstract tasks, treated as primitive at this level, hold during sequential execution from the initial state.*

Definition 9 (Planning Process Interface)**.** *A* planning process *(PP) is a 5-tuple $(D, I, \ell, \mathcal{I}, \mathcal{F})$ where:*

- $D = (T_a, T_p, M, L)$ *is the planning domain*
- $I \in I_S$ *is the internal state of the planning process.* I_S *is the set of all internal states and this depends on the concrete implementation.*
- $\ell \in \{1, ..., L\}$ *is the current detail level*
- $\mathcal{I} : I_S \times \mathcal{P}(A) \to I_S \times (T_a \cup T_S \cup T_p \cup \{null\})$
  Given the current internal state and the world state, this function returns the updated internal state and either the next task to be added to the plan, or null to indicate failure.
- $\mathcal{F} : I_S \times \mathcal{P}(A) \times T_S \to I_S \times \mathbb{N}$
  Handles the failure of a previously generated separable task $t \in T_S$ *under the given world state. Returns the updated internal state and the number of previously generated tasks to retract (i.e., undo). The next call to* $\mathcal{I}$ *will generate a new alternative task.*

Internally, depending on the implementation, $\mathcal{F}$ will trigger backtracking or replanning. After $\mathcal{F}$ is called, the next call to $\mathcal{I}$ will return either a new task to replace the failed one or NULL if replanning fails entirely.

Implementation Notes. This interface can be realized through different search strategies:

- *MCTS-based*: Uses Monte Carlo tree search where backtracking simply means that we go back and select a task that has a smaller visit count than the discarded one.
- *HTN with repair*: Rolls back the currently failed plan, discards the failed task, and then attempts to complete a new plan by reusing parts of the previously failed one. [24].

Example In the city example, a planning process generates a plan to construct the buildings at the highest detail level 3 when given a quarter task as input.

Relationship to classical HTN planning. If the domain provides only one level ($L = 1$) and contains no separable abstract tasks, HWSP degenerates into standard HTN planning: the single PP is invoked exactly once, no spawning occurs, and the algorithm behaves identically to a flat HTN search.

3.4. Planning Process Management

The HWSP algorithm manages planning processes through a stack-based approach. The operational flow proceeds as follows:

1. **Creation:** When encountering a separable abstract task, a new planning process is created at the next detail level, with the effects of the separable task as its goals.
2. **Execution:** Only the topmost process on the stack is active, producing tasks that are either added to the main plan (if primitive) or used to spawn new processes (if separable abstract task).
3. **Termination:** A process terminates upon achieving its goals or failing. On success, its parent process resumes; on failure, backtracking occurs.

The main planning algorithm initializes with the first planning process at detail level one, an empty plan, and the initial world state. While the stack contains processes, the algorithm retrieves the next task from the topmost process. If the process is finished, it is removed from the stack. If a process fails to produce a task, the algorithm backtracks according to the domain class rules. For primitive tasks, the algorithm adds them to the plan and updates the world state with their effects. For separable abstract tasks, it creates a new planning process and pushes it onto the stack. The algorithm terminates successfully when the goal is achieved, or with failure when the stack becomes empty without achieving the goal.

Definition 10 (Overall Solution)**.** *A solution to the overall planning problem* $\Pi = (s_0, t_0)$ *is a plan* Q *where: (1) every leaf node is a primitive task (*$\lambda_T(v) \in T_p$*), and (2) all primitive task preconditions hold during sequential execution from* s_0*.*

Example In the city domain, the main planning algorithm would be coordinating the planning process so that a complete city is built.

As demonstrated in Figure 2, when a planning process encounters a failure, the backtracking behavior is determined by the domain class.

State projection restricts each planning process to atoms at the appropriate detail level, reducing memory usage. The behavior of HWSP is further influenced by domain-specific properties, which we classify into EX1–EX3 in the Section 4.

4. Domain Classes

The performance of planning algorithms, such as Hierarchical Task Network (HTN) planning and its extensions like Hierarchical World State Planning (HWSP), can be significantly influenced by the structure of the planning domain. This observation aligns with the downward refinement property introduced by Bacchus and Yang [25], which emphasizes how domain structure impacts the feasibility and efficiency of hierarchical refinement. To formalize these influences in the context of separable abstract tasks, we introduce a classification of domain classes based on task interaction and backtracking behavior. In particular, we analyze how sibling tasks, parallel subtasks at the same abstraction level, may or may not interfere with one another.

Let's consider our urban planning example again, where a city manager needs to plan the construction of new buildings in different quarters of the city. To demonstrate the algorithm's ability to handle diverse domain scenarios, we modify this base domain in three key ways:

- First, we introduce *blocked cells*, which may cause tasks to fail and lead to different planning outcomes.
- Second, we enable *electricity constraints*, where a single `powerplant`—placed in any quarter—can power the entire city.
- Third, *power propagation* can be expressed using derived predicates D_{der}: a quarter is considered powered if its city contains a powerplant, captured by $\mathsf{has\_power}(\mathsf{quarter})$.

These modifications highlight distinct domain classes, such as spatial constraints, resource allocation, and hierarchical dependencies. We propose several domain classes, which we will describe using illustrative examples related to placing buildings in a quarter:

1. **EX1: Execution Guaranteed**
   Intuitive Definition: Execution (decomposition of the separable abstract task) is guaranteed once preconditions are fulfilled.
   Example: In our urban planning scenario, if a quarter plan fits (i.e., the preconditions of the separable abstract task are fulfilled), its construction is guaranteed without any unforeseen interference from other quarters or tasks.

   Definition 11 (Domain class EX1)**.** *A domain D is in **EX1** iff for every plan Q and every separable task occurrence $v \in V_T$ of Q, if* $\mathsf{is\_sep}(Q, v) \wedge$ *preconditions of* $\lambda_T(v)$ *are satisfied in* $\mathsf{state\_at}(Q, v)$*, then* $\mathsf{can\_decompose}(Q, v) =$ *true.*

2. **EX2: Execution Can Fail but Won't Benefit from Backtracking into Siblings**
   Intuitive Definition: Execution can fail, but a different solution from another planning process on the same detail level won't help.
   Example: In a more realistic urban planning scenario, parts of the land within a quarter might not be buildable due to unforeseen reasons (e.g., hidden environmental hazards). As a result, the construction of a specific building – and possibly the entire quarter – may fail, even if all preconditions appeared to be satisfied. However, changing the plan for a different quarter (i.e., a sibling task) will not influence the outcome. If such a failure occurs, the planner must backtrack to the parent task (e.g., the city-level plan) and select a new decomposition that avoids

the problematic quarter altogether. The key EX2 property is that the decomposability of one separable abstract task (e.g., a quarter) is independent of the concrete plans chosen for earlier or parallel sibling tasks.

Definition 12 (Domain class EX2)**.** *D is in **EX2** iff for all plans Q, separable occurrences $v \in V_T$ and for all sibling abstract task occurrences s with the same parent as v in the plan's strict total order $\prec$ over task nodes with $L_d(s) = L_d(v)$,* $\mathsf{is\_sep}(Q, s) \wedge \mathsf{is\_sep}(Q, v) \wedge Q \rightsquigarrow_s Q' \Longrightarrow \mathsf{can\_decompose}(Q, v) = \mathsf{can\_decompose}(Q', v)$.

In words, replacing the internal subtree of an earlier separable sibling s does not affect the decomposability of a later separable task at the same detail level.

3. **EX3: Execution Can Fail and May Benefit from Backtracking**
   Intuitive Definition: Execution can fail, and another solution from a prior planning process might help.
   Example: In a complex urban planning scenario, the placement of a building in one quarter could influence the feasibility of constructing buildings in adjacent quarters due to shared resources or spatial constraints. If the construction of a building fails because there is no power, creating an industrial quarter with a power plant prior will help.

   Definition 13 (Domain class EX3)**.** *The class **EX3** includes all planning domains.*

   It represents the general case where no specific assumptions about separable task independence are made. Note that introducing derived predicates such as $\mathsf{has\_power}(\mathsf{quarter})$ can explicitly model task dependencies, allowing EX3 domains to be reclassified as EX2 if sibling tasks become independent under these conditions.

Separable abstract tasks isolate planning to local *islands* of the world state. How strongly such islands interact determines how much global backtracking remains necessary. We capture this idea by three *domain classes*, EX1 – EX3, that impose increasingly weak independence assumptions on separable tasks.

Proposition 1 (Independence chain)**.** *EX1 $\subset$ EX2 $\subset$ EX3.*

Proof. Immediate from the definitions: EX1 implies the EX2 invariance because the antecedent of Def. 12 is stronger; conversely any counter-example to EX2 is by definition a witness for EX3. □

5. Backtracking

In Hierarchical World State Planning (HWSP), backtracking plays a pivotal role due to its unique approach to task decomposition and execution. While HWSP shares similarities with traditional Hierarchical Task Network (HTN) planning, its backtracking mechanism in the main planning system is distinct.

In HWSP, backtracking occurs when a planning process fails to find a viable solution. At this point, the algorithm must reconsider its previous decisions and explore alternative paths. This is where the domain classes (EX1, EX2, and EX3) come into play, as they significantly influence the backtracking behavior.

- **EX1**: In domains where execution is guaranteed once preconditions of a separable abstract task are met, backtracking is relatively straightforward. Since each task's success is assured, the main planner never needs to backtrack.
- **EX2**: In EX2 domains, the failure of a separable abstract task requires specific backtracking behavior. Let us analyze why EX2 leads to this behavior through a logical proof.

  Proposition 2 (EX2 Backtracking Behavior)**.** *In an EX2 domain, if a separable abstract task v fails during decomposition, the planner must backtrack to the parent planning process without considering siblings of v.*

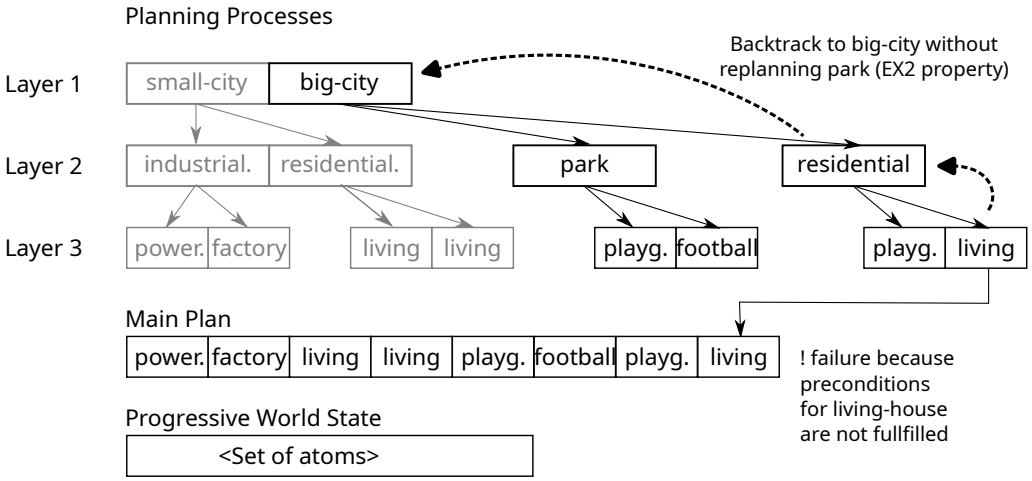

Figure 2: When decomposing the second city (big-city), the first quarter (park) succeeds, but the second quarter (residential-quarter) fails. Since EX2 guarantees that earlier siblings (like park) cannot influence the feasibility of later tasks, the planner does not attempt to replan the park. Instead, it backtracks to the parent (big-city) and selects a new decomposition – this time choosing its first child to be an industrial-quarter (which will provide electricity) instead of a park. This demonstrates that backtracking in EX2 must go up to the parent level, not across to siblings. We use the following abbreviations: industrial. = industrial-quarter, residential = residential-quarter, playg. = playground, power. = powerplant, football = football field.

Proof. By Definition 12, for any plan Q and separable task v, modifying any sibling s (with $s \prec v$) does not affect the decomposability of v. Formally: $\mathsf{can\_decompose}(Q, v) = \mathsf{can\_decompose}(Q', v)$ where $Q \rightsquigarrow_s Q'$.
If v fails ($\mathsf{can\_decompose}(Q, v) = \textsc{false}$), altering any prior sibling s through backtracking cannot make $\mathsf{can\_decompose}(Q', v) = \textsc{true}$. Thus, the only recourse is to inform the parent process of v's failure. The parent must then generate a new separable abstract task to replace v or one of its siblings. If the parent exhausts all alternatives and fails, the failure propagates upward recursively. This eliminates the need to explore siblings of v, as their modifications cannot resolve v's failure under EX2's independence. The algorithm remains sound and complete [20]. □

- **EX3**: In these domains, where execution can fail and backtracking may help, we must perform backtracking as in the original HWSP algorithm [1].

6. Results

This section establishes formal upper bounds on the search effort of *Hierarchical World-State Planning* (HWSP) when executed on the three domain classes EX1 ⊂ EX2 ⊂ EX3 and contrasts those bounds with classical HTN planning. All logarithms are base 2; plan-existence complexities follow the taxonomy of Erol et al. [22].

6.1. Complexity model

Consider a fixed detail level ℓ. Let n_ℓ denote the number of *sibling* separable task occurrences emitted at that level, and let each sibling have at most k_ℓ decomposable methods. The refinement of *one* sibling induces a local search tree of branching factor $b_{\ell+1}$ and depth $h_{\ell+1}$; we abbreviate $B_{\ell+1} = b_{\ell+1}^{h_{\ell+1}}$.

Let N denote the total input size (i.e., the size of the domain and the initial task network). We assume that $n_\ell = \mathcal{O}(N)$; that is, the number of sibling tasks at each level grows at most linearly with the input. The analysis that follows is worst-case and *tight*: for every bound, we can construct an infinite family of instances whose planners visit exactly the stated number of states.

We analyze the number of states visited during search, denoted as:

- S_{HTN}: states visited by classical HTN planning
- S_{EX1}, S_{EX2}, S_{EX3}: states visited by HWSP in the respective domain classes

6.2. State-space bounds

Theorem 3 (EX3 versus HTN). *For any level* ℓ

$$S_{\mathrm{HTN}} = B_{\ell+1}^{n_\ell}, S_{\mathrm{EX3}} = n_\ell \, k_\ell^{n_\ell} \, B_{\ell+1}, \frac{S_{\mathrm{HTN}}}{S_{\mathrm{EX3}}} = \Theta\left(\frac{B_{\ell+1}^{n_\ell - 1}}{n_\ell \, k_\ell^{n_\ell}}\right)$$

Proof. Classical HTN planning *inlines* the local search of each sibling.[1] Consequently the global search tree contains $B_{\ell+1}$ possibilities for the first sibling, $B_{\ell+1}$ for the second, and so on, for a total of $B_{\ell+1}^{n_\ell}$ states.

In EX3, the parent process still explores every combination of high-level alternatives, hence $k_\ell^{n_\ell}$ parent choices. However, once such a combination has been chosen, each sibling is solved *independently* by its own local search; the parent never needs to interleave those searches. The resulting state count is therefore $n_\ell \, k_\ell^{n_\ell} \, B_{\ell+1}$. Taking the quotient of the two expressions yields the claimed factor. □

Theorem 4 (EX2 versus EX3). *For any level* ℓ

$$S_{\mathrm{EX2}} = n_\ell \, k_\ell \, B_{\ell+1}, \qquad \frac{S_{\mathrm{EX3}}}{S_{\mathrm{EX2}}} = \Theta\left(\frac{k_\ell^{n_\ell - 1}}{n_\ell}\right)$$

Proof. Definition 12 states that substituting an *earlier* sibling by an alternative decomposition never changes the decomposability of a *later* sibling. Hence, when a sibling fails, the planner backtracks exactly one level up, replaces *only* that sibling, and leaves all others untouched. Worst case it will test every one of the k_ℓ methods before a success (or final failure) is found. Since there are n_ℓ siblings, at most $n_\ell \, k_\ell$ parent alternatives are generated, each of which spawns one local search of size $B_{\ell+1}$. The product yields S_{EX2}. Dividing by S_{EX3} establishes the gap. □

The bounds given in Theorems 3 and 4 are tight in the following sense: for each setting of n_ℓ, k_ℓ, and $B_{\ell+1}$, there exists a family of domains and initial task networks for which the number of visited states matches the bound asymptotically, i.e., $S = \Theta(\cdot).\Theta(S)$ states.

HTN Tightness Example. Let the initial task be a sequence of n_ℓ abstract tasks, each with k_ℓ methods, where each method expands to a local plan space of size $B_{\ell+1}$ (e.g., a binary tree of depth $\log_2 B_{\ell+1}$). If no heuristic guidance is used, the planner must exhaustively search all $B_{\ell+1}^{n_\ell}$ combinations.

EX3 Tightness Example. Consider a domain with n_ℓ sibling tasks (e.g., city quarters). Each has k_ℓ decomposition methods, exactly one of which builds a `power plant`. Suppose that every quarter requires electricity to decompose, and power flows only from left to right: quarter q_j can be decomposed only if some earlier sibling q_i with $i < j$ contains a `power plant`. This induces a causal dependency chain across siblings: choosing non-power plant methods early may prevent later decompositions, forcing backtracking and revision of earlier choices. In the worst case, it explores all $k_\ell^{n_\ell}$ global method assignments; for each one, it performs n_ℓ independent subsearches of size $B_{\ell+1}$. The total number of visited states is therefore $n_\ell \, k_\ell^{n_\ell} \, B_{\ell+1}$, which matches the upper bound for EX3. The cross-sibling dependency violates the EX2 invariance condition (Definition 12), so this domain lies strictly in EX3.

Thus, the upper bound for S_{EX3} is *tight*, and the factor of n_ℓ is necessary in all general EX3 domains where sibling tasks do not share causal links.

[1] We assume classical HTN planning operates under the same restrictions as HWSP: acyclic networks, progressive decomposition, and bounded depth, ensuring a fair comparison.

EX2 Tightness Example. Construct a domain with n_ℓ separable sibling tasks (e.g., one per city quarter), each having k_ℓ distinct task alternatives. For each sibling, only one of its k_ℓ alternatives is decomposable, while the others are designed to fail due to local, hidden constraints (e.g., unobservable predicates or infeasible layouts). These constraints are specific to each sibling and cannot be inferred or affected by the other siblings' choices. When a sibling task fails, the parent planning process must replace it with another alternative, up to k_ℓ times in the worst case. Each attempted task triggers a local planning process that explores a subsearch space of size $B_{\ell+1}$. Since all sibling tasks operate over disjoint parts of the world state and have no shared predicates, changing one sibling's decomposition does not influence the decomposability of another—satisfying the EX2 invariance condition (Definition 12). The total number of visited states in this construction is $n_\ell \cdot k_\ell \cdot B_{\ell+1}$, matching the upper bound for EX2 domains and proving its tightness.

On EX1. EX1 guarantees *downward refinement* for the *root* plan: once the preconditions of a separable task are satisfied, its decomposition cannot fail. Inside the local planning process spawned for that task, however, non-monotonic search (e.g. local backtracking) is still permitted. Therefore EX1 improves the constant factors of Theorem 4 but does not change its asymptotic form. More precisely, the number of search states in EX1 is $S_{\mathrm{EX1}} = n_\ell \, B_{\ell+1}$, as each of the n_ℓ separable tasks is successfully decomposed on the first attempt, requiring a single local search of size $B_{\ell+1}$ per task.

Corollary 5 (Strict inclusion chain)**.** *It holds $S_{\mathrm{EX1}} < S_{\mathrm{EX2}} < S_{\mathrm{EX3}} < S_{\mathrm{HTN}}$ as each "<" removes one exponential factor in the input size N, assuming $n_\ell = O(N)$.*

6.3. Connection to classical complexity theory

Bacchus and Yang's *Downward Refinement Property* (DRP) [25] formalizes a key structural criterion for efficient hierarchical planning: once a high-level plan is found, it can be refined monotonically without modifying earlier decisions. Our EX1 domain class satisfies DRP by construction, as successful decomposition is guaranteed once preconditions are met.

EX2 domains do not satisfy DRP in the strict sense, since the planner may backtrack and replace earlier siblings. However, they enforce a weaker property we term *internal decomposition invariance*: the feasibility of decomposing a task is invariant under changes to the internal decomposition of prior siblings at the same abstraction level. This allows EX2 domains to retain many of the practical benefits of DRP, such as reduced backtracking and localized refinement, but without full top-down monotonicity.

Korf's macro-operator theorem [5] predicts an exponential-to-linear collapse in ideal DRP-compatible settings, which applies fully to EX1 and partially to EX2. Erol et al. [22] showed that plan existence is *semi-decidable* for unrestricted HTNs but becomes PSPACE-complete under acyclic, non-interleaved constraints—the same assumptions that characterize our EX3 class. Structural restrictions like those in EX2 further reduce the search space, pushing the complexity down to NP-complete or even as low as P in well-structured domains.

Here, N is the combined size of the initial task network and domain description; p and c are fixed polynomials.

6.4. Effect of a (fallible) binary predictor

In HWSP every separable task t_S comes with a binary *decomposition predictor* $\sigma : \mathcal{P}(A) \times T_S \to \{\textsc{true}, \textsc{false}\}$. Intuitively, $\sigma(s, t_S) = \textsc{true}$ says that local planning *ought* to succeed.[2] Mis-predictions are the only reason why an EX2 planner ever needs to backtrack.

Error model. For the sake of analysis we assume that, *conditioned on the current state*, the predictor returns the correct answer with probability p (*accuracy*) and the wrong answer with probability $\bar{p} = 1 - p$.

[2] We treat σ as a black box: it may be a learned classifier, a logical test or a mix thereof.

Theorem 6 (Expected search effort with a fallible predictor)**.** *Let $S^{\star}_{\mathrm{EX2}} = n_\ell B_{\ell+1}$ be the number of search states when the predictor is* perfect*. Assume that, each time a separable task is encountered, the predictor is correct with probability p and wrong with probability $\bar{p} = 1 - p$ (independently of previous calls). If the predictor errs, the parent process tries a different alternative; at most $k_\ell - 1$ further alternatives exist.*

Then the expected number of visited states is bounded by $\mathbb{E}[S_{\mathrm{EX2}}(p)] \;\le\; S^{\star}_{\mathrm{EX2}} \left(1 + \bar{p}\,(k_\ell - 1)\right)$.

Proof. Let n_ℓ be the number of separable task occurrences at level ℓ, and let each task require a local search of size $B_{\ell+1}$ once a valid decomposition is selected. Under a perfect predictor, each task is decomposed successfully on the first try, so the total number of visited states is: $S^{\star}_{\mathrm{EX2}} = n_\ell \cdot B_{\ell+1}$.. Now consider a fallible predictor with accuracy p, and let $\bar{p} = 1 - p$ denote the probability of an incorrect prediction. If the predictor fails (e.g., produces a false positive), the planner must backtrack and try a different decomposition. Since each task has at most k_ℓ possible alternatives, the number of additional alternatives to try after a misprediction is at most $k_\ell - 1$.

Thus, the expected number of task attempts for each separable task is at most: $1 \cdot p + (1 + \beta) \cdot \bar{p}, \quad$ with $\beta \le k_\ell - 1$. Substituting, we obtain the upper bound: Expected attempts per task $\le 1 + \bar{p} \cdot \beta \le 1 + \bar{p} \cdot (k_\ell - 1)$. Multiplying this overhead by the base cost $S^{\star}_{\mathrm{EX2}}$ yields: $\mathbb{E}\left[S_{\mathrm{EX2}}(p)\right] \le S^{\star}_{\mathrm{EX2}} \cdot (1 + \bar{p} \cdot (k_\ell - 1))$, which proves the claim. □

A perfect predictor collapses EX2 to EX1. Setting $p = 1$ makes the overhead term $(1 + \bar{p}\,\beta)$ equal to 1, so every separable task is decomposed successfully on the very first try. This eliminates *all* global backtracking and re-establishes the Downward-Refinement Property for the top-level plan – precisely the hallmark of the EX1 class (Def. 11). Formally, the planner's behavior becomes observationally equivalent to running HWSP on an EX1 domain. □

Interpretation. Theorem 6 shows that the predictor's accuracy enters the search complexity as a *linear* scalar factor. Even modest accuracies (say $p = 0.9$) leave the $\Theta\big(k_\ell^{n_\ell - 1}\big)$ gap of Theorem 4 largely intact, because $\bar{p} \le 0.1$. Only adversarially bad predictors ($p \to 0$) would degrade EX2 back to EX3 behaviour.

7. Conclusion

We analyzed Hierarchical World State Planning (HWSP), a planning framework that extends classical HTN planning through multi-level state abstraction and separable abstract tasks. While HWSP was introduced in prior work, its theoretical properties and complexity characteristics had not been formally studied.

Our main contribution is a comprehensive theoretical analysis of HWSP's planning dynamics. We introduced a novel classification of planning domains, EX1, EX2, and EX3, based on structural constraints on separable abstract tasks and their backtracking behavior. This classification enables a deeper understanding of when HWSP achieves substantial reductions in search complexity compared to classical HTN planning.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT and LLaMA 3.3 70B in order to: grammar and spelling check. After using these tools, the author(s) reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Staud, Integrating Deep Learning Techniques into Hierarchical Task Planning for Effect and Heuristic Predictions in 2D Domains, in: HPLAN Workshop, 2023.

[2] R. E. Korf, Planning as Search: A Quantitative Approach, AI 87 33 (1987) 65–88.

[3] C. A. Knoblock, Learning Abstraction Hierarchies for Problem Solving, in: AAAI, 1990, pp. 923–928.

[4] R. I. Brafman, C. Domshlak, Factored Planning: How, when, and When Not, in: AAAI, volume 6, 2006, pp. 809–814.

[5] R. E. Korf, Macro-Operators: A Weak Method for Learning, Artificial Intelligence 26 (1985) 35–77.

[6] A. Botea, M. Enzenberger, M. Müller, J. Schaeffer, Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators, Journal of Artificial Intelligence Research 24 (2005) 581–621.

[7] T. G. Dietterich, Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition, Journal of Artificial Intelligence Research 13 (2000) 227–303.

[8] T. D. Kulkarni, K. Narasimhan, A. Saeedi, J. Tenenbaum, Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation, Advances in Neural Information Processing Systems 29 (2016).

[9] D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, S. Mitchell, Total-Order Planning with Partially Ordered Subtasks, in: IJCAI, volume 1, 2001, pp. 425–430.

[10] C. Olz, P. Bercher, A Look-Ahead Technique for Search-Based HTN Planning: Reducing the Branching Factor by Identifying Inevitable Task Refinements, in: Proceedings of the International Symposium on Combinatorial Search, volume 16, 2023, pp. 65–73.

[11] R. Tsuneto, K. Erol, J. Hendler, D. Nau, Commitment strategies in hierarchical task network planning, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 1996, pp. 536–542.

[12] R. Alford, P. Bercher, D. Aha, Tight Bounds for HTN Planning, in: Proceedings of the International Conference on Automated Planning and Scheduling, volume 25, 2015, pp. 7–15.

[13] T. Geier, P. Bercher, On the Decidability of HTN Planning with Task Insertion, in: IJCAI, 2011, pp. 1955–1961.

[14] T. L. McCluskey, Object Transition Sequences: A New Form of Abstraction for HTN Planners, in: AIPS, 2000, pp. 216–225.

[15] E. D. Sacerdoti, Planning in a Hierarchy of Abstraction Spaces, Artificial intelligence 5 (1974) 115–135.

[16] C. A. Knoblock, Automatically Generating Abstractions for Planning, Artificial Intelligence 68 (1994) 243–302.

[17] B. Marthi, S. Russell, J. A. Wolfe, Angelic Hierarchical Planning: Optimal and Online Algorithms, in: ICAPS, 2008, pp. 222–231.

[18] G. Behnke, D. Höller, S. Biundo, Finding optimal solutions in htn planning-a sat-based approach., in: IJCAI, 2019, pp. 5500–5508.

[19] D. Höller, G. Behnke, P. Bercher, S. Biundo, H. Fiorino, D. Pellier, R. Alford, HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems, in: Proc. of the AAAI Conference on AI, volume 34, 2020, pp. 9883–9891.

[20] M. Staud, Urban Modeling via Hierarchical Task Network Planning, in: HPLAN Workshop, 2022.

[21] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, Elsevier, 2004.

[22] K. Erol, J. Hendler, D. S. Nau, Complexity Results for HTN Planning, Annals of Mathematics and Artificial Intelligence 18 (1996) 69–93.

[23] S. Edelkamp, J. Hoffmann, PDDL 2.2: The Language for the Classical Part of IPC-4, in: International Planning Competition, 2004.

[24] D. Höller, P. Bercher, G. Behnke, S. Biundo, HTN Plan Repair Using Unmodified Planning Systems, in: HPLAN (ICAPS), 2018, pp. 26–30.

[25] F. Bacchus, Q. Yang, The Downward Refinement Property, in: IJCAI, 1991, pp. 286–293.